VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozšíření frameworku pro animaci algoritmů

Extension of Automated animation framework

2009

Jiří Chmiel

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1.5.2009 .....

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Davidovi Ježkovi za jeho podnětné návrhy a čas, který mi věnoval.

Abstrakt

Práce se zaobírá analýzou, návrhem a vývojem aplikačního rámce umožňujícího jednoduchou definici zdrojového kódu algoritmu napsaného v programovacím jazyce Java a jeho následnou automatickou vizualizaci. Při návrhu budu klást důraz na jednoduchost použití a hlavně znovupoužitelnost. Popisovat budu hlavně myšlenky ohledně definice a provádění kódu a jeho provázanosti s vizualizačními prvky. Poté budu popisovat jeho následnou implementaci. Vyvíjený aplikační rámec bude účinným nástrojem na tvorbu pomůcek, ktere umožní vyučujícím efektivněji vysvětlovat a studentům snáze chápat principy základních algoritmů používaných v informatice.

Klíčová slova: aplikační rámec, algoritmus, Java, Swing, vizualizace

Abstract

This work is concerned with analysis, design and development of an aplication framework enabling a simple definition of the source code of the algorithm, written in the Java language, and its follow-up automatic visualization. When designing, I'll accentuate on simplicity of using and possibility of reuse. I'll mainly describe the ideas about definition and execution of the code and its interconnection with visual elements. After that I'll describe its follow-up implementation. This aplication framework will be an effective tool for creation of the teaching materials, which will allow effective teaching and easier understanding of principles of the basic algorithms used in computer science.

Keywords: framework, algorithm Java, Swing, visulization

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface, rozhraní pro programování aplikací

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Každý z nás se již určitě snažil někdy něco někomu vysvětlit a každému již určitě bylo někdy něco vysvětlováno. Vysvětlovat nebo pochopit složitou myšlenku či postup není vůbec nic jednoduchého, zvláště máme-li k dispozici pouze slova. Situace se zlepší můžeme-li problém načrtnout. Obrázek prostě vydá za tisíce slov. Mezi postupy obzvláště složité na vysvětlování patří počítačové algoritmy. Počítačový algoritmus lze zapsat například ve zdrojovém kódu. Tento kód však není člověku moc srozumitelný. Člověku je mnohem stravitelnější vidí-li průběh algoritmu na obrázku, nebo ještě lépe, v animaci.

Nástrojů určených k vytváření animací existuje spousta od nějakého PowerPointu až po sofistikovaný Flash. Všechny tyto nástroje mají jednu nevýhodu a tou je znovupoužitelnost. Bude-li někdo chtít vysvětlit princip například bublinkového třídění, vytvoří si animaci, bude-li však později chtít vysvětlit princip třídění QuickSort musí si vytvořit animaci novou, což stojí čas a námahu. Mnohem pohodlnější by bylo, kdyby byl po ruce nástroj, který by dokázal z nesrozumitelného zdrojového kódu automaticky nebo z malou autorovou námahou vytvořit srozumitelnou a všeříkající animaci. Určitá účast člověka je samozřejmě nutná, protože těžko půjde vyvinout nástroj, který by ze zdrojového kódu dokázal sám vyčíhnout, která část algoritmu je pro pochopení principu důležitá, a navrhnout, jakým způsobem jí načrtnout. Nicméně se tomuto procesu dá hodně pomoci.

Cílem mé bakalářské práce je navázat a rozšířit na loňskou bakalářskou práci Evy Plackové [4]. Jedná se o návrh a implementaci aplikačního rámce umožňující sestavení zdrojového kódu, simulování chování počítače při jeho vykonávání a následně prezentování dějů uvnitř simulovaného počítače. Při návrhu budu klást důraz na použitelnost a rozšiřitelnost. A tyto vlastnosti budu poté demonstrovat na praktických příkladech.

2 Specifikace požadavků

Celý aplikační rámec je určen pro aplikace psané v jazyce Java. Proto i prezentované algoritmy, budou zapsány v tomto jazyce. Cílem je minimální snaha autora-programátora, proto všechny vazby mezi vykonáváním kódu algoritmu a jeho prezentací musí být připraveny s případnou možností jednoduchého rozšíření.

2.1 Požadavky na funkčnost

Aplikační rámec musí být schopen nasimulovat a znázornit většinu operací (všechny mimo ternární ? : a přiřazovacích operací +=,−= ...), musí umět pracovat s proměnnými, poli a triviálními funkcemi (triviální funkce je taková, jejíž průběh nebude znázorňován, půjde jen o zavolání nějaké jednoduché vnitřní funkce např. random()). Proměnné se musí ukládat hodnotou i odkazem. Dále musí umět volat (i rekurzivně) jiné uživatelem definované funkce a úspěšně se z nich vracet. Umět nasimulovat chování základních programových bloků (**if**−**else** , **do**−**while**, **while**−**do**, **for**), programový blok **switch**−**case** použit neubude. Aplikační rámec bude umět nasimulovat i chování jednoduchých tříd s veřejnými atributy, avšak plně postihnout celou problematiku objektového programování ne. Dále není v zadání implementovat možností výčtových prostředí, vyjímek či generik. To proto, že tyto jazykové elementy většinou nebývají základním kamenem algoritmů.

2.2 Požadavky na zápis algoritmu

K definici zdrojového kódu prezentovaného algoritmu bude sloužit jednoduché intuitivní programové rozhraní. Do budoucna by se pak mohlo do implementovat načítání z externích zdrojů (např. textového souboru).

2.3 Požadavky na animaci

Mezi základní požadavky na animace patří snadná zakomponovatelnost do Java Aplikací popřípadě Java Appletů pracujícími s grafickými komponentami JavaSwing[2]. Animace musí být plynulá a nesmí způsobovat „zamrzání“ ostatních prvků grafického uživatelského rozhraní; předpokládá se tedy použití vláken. V aplikačním rámci budou k dispozici základní animační entity pro znázornění proměnné a pole, musí být také umožněno vkládat jiné komponenty technologie JavaSwing. Použití animačních entit musí být intuitivní z ne příliš složitým API. Animační entity budou dále automaticky reagovat na události vyvolané vykonáváním prezentovaného algoritmu.

3 Analýza aplikačního rámce

Kompletní třídní diagram aplikačního rámce je znázorněn v příloze D. Cely rámec je rozdělen na čtyři hlavní logické části. Na část výkonou (balíky stroj3 a stroj3.strom), na část kódů (balík kody), na část elementů jazyka (balík jayzk) a poslední částí je část vizualizační (balík prezentace).

- Část výkonná má na starosti řízení vykonávání kódu, ukládání proměnných do paměti, vyhodnocování výrazů a přeposílání zpráv o událostech během vykonávání algoritmu.
- Část sestavení kódu obsahuje třídy a funkce pro definici kódu zobrazovaného algoritmu a také pro jeho vykonávání. Jsou zde implementovány logiky základních programových bloků **if**−**else** , **do**−**while**, **while**−**do**, **for**, volání funkcí apod.
- V části jazykových elementů jsou definovány základní datové typy, operace, triviální funkce, a použité symbolické odkazy.
- Poslední vizualizační část má na starosti výslednou vizualizaci.

Všechny čtyři části podrobně rozepíšu v následujících kapitolách.

Nad celým aplikačním rámcem pracuje klientská aplikace, která si nejprve pomocí jazykových elementů dostupných ve třídě jazyk.Jazyk a pomocí tříd kody.KodFunkce a kody.KodProgramu nadefinuje zdrojový kód prezentovaného algoritmu. Poté co je kód algoritmu hotov vytvoří se jeho prezentace (např. které proměnné se budou zobrazovat, jejich vzhled, komentáře apod.). Nadefinovaný kód se předá částí výkoné (třída stroj3.Stroj), kde jej lze krokovat po jednotlivých příkazech. Během tohoto krokování se generují příslušné události na které reaguje část vizualizační, která vše vizualizuje.

4 Návrh a implementace výkonné části

Část výkonná se dělí na část řízení kódu, která poskytuje rozhraní pro nahrání, spouštění a krokování algoritmu, část generování událostí, a na podpůrné části vyhodnocování výrazů a paměť. Dále je zde definována řada rozhraní definující základní jazykové prvky.

4.1 Implementace řízení kódu a generování událostí

Třídní diagram je znázorněn na obrázku 1. Základem je Třída Stroj, která je implementována jako jedináček (dle návrhového vzoru Singleton). Instanci lze pak získat statickou metodou dejStroj(). Třída Stroj má vazbu na objekt rozhraní IKodProgramu, toto rozhraní reprezentuje programový kód prezentovaného algoritmu. Kromě něj třída Stroj obsahuje posluchače událostí reprezentovaného vnitřní třídou StrojObsluhaUdalosti. Tato třída přeposílá přijímané události všem posluchačům připojeným ke stroji.

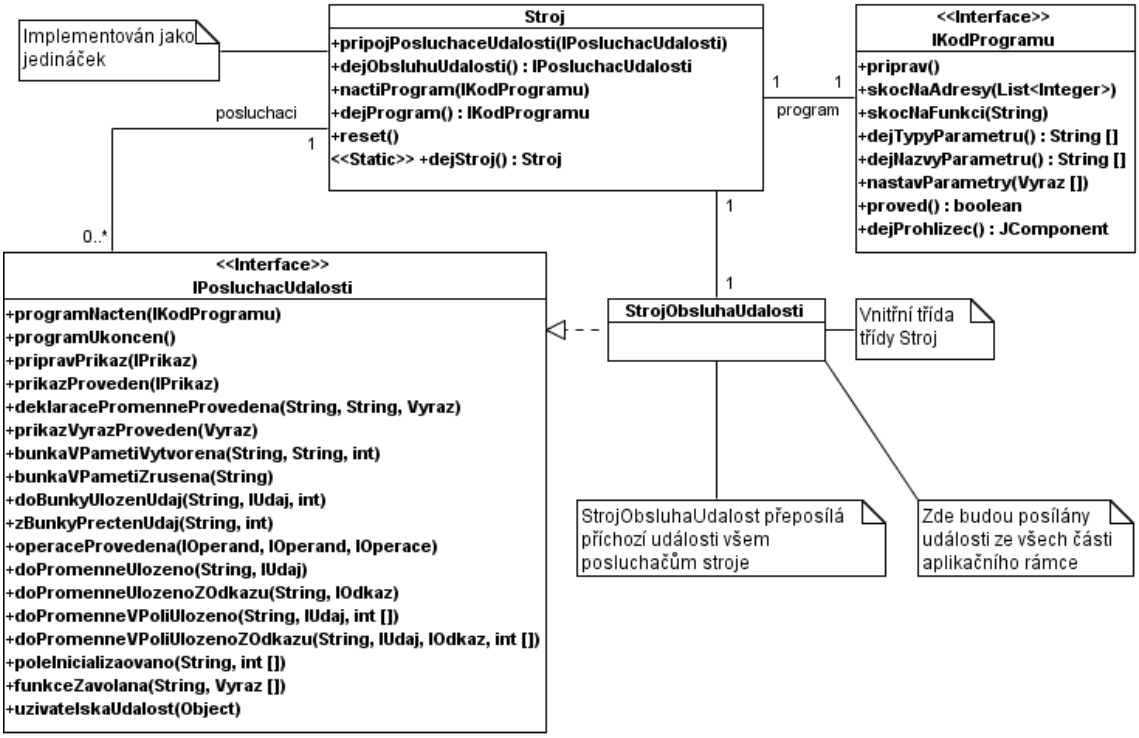
Třída Stroj obsahuje několik metod sloužící jako rozhraní pro řízení kódu. Pomocí metody nactiProgram(IKodProgramu) lze do stroje nahrát kód příslušného algoritmu, zpětně jej lze pak získat metodou dejProgram(). Obsluhu událostí lze získat pomocí metody dejObsluhuUdalosti(),připojit posluchače potom metodou pripojPosluchaceUdalosti(). Přerušit krokování programu a přivést stroj do výchozího stavu lze metodou reset().

Prezentovaný algoritmus lze krokovat metodou proved() definovanou v rozhraní IKodProgramu, tato metoda vrací logickou hodnotu. Je-li vrácená hodnota **true** znamená to, že algoritmus dospěl do svého konce a krokování je již ukončeno, je-li vráceno **false** algoritmus stále běží.

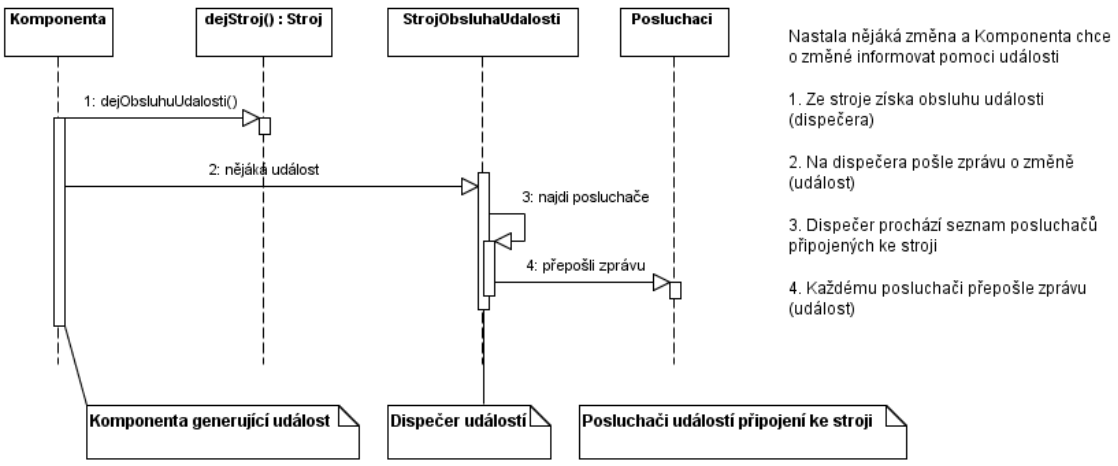
Aby mohla vizualizace probíhat korektně je nutné všechny její komponenty informovat o změnách, které nastaly v souvislosti s prováděním prezentovaného algoritmu. Mezi takové změny patří např. načtení kódu algoritmu, uložení hodnoty do proměnné, volání funkce apod. Je vhodné aby existovalo jedno místo, dispečer, kam by všechny části mohly posílat informace o právě vzniklých změnách a kam by se také mohly připojit všechny části na tyto změny reagující. Obsluha událostí ve třídě Stroj pracuje v roli dispečera, který přeposílá příchozí zprávy všem ostatním posluchačům stroje. Mechanismus je znázorněn na obr. 2 Komponenta mající potřebu vyvolat událost ze stroje, který je v systému pouze jeden, získá obsluhu událostí na kterou pošle zprávu s příslušnými parametry. Obsluha události ve třídě Stroj potom tuto zprávu přepošle všem posluchačům připojeným ke stroji. Takovým posluchačem může být například vizualizační komponenta. Všechny událostí jsou zobrazeny v tabulce 1. Mezi standardními událostmi se nachází i uživatelská událost, která může být využita při rozšiřování aplikačního rámce o další prvky.

4.2 Rozhraní definující základní jazykové prvky

V balíku stroj3 je také definováno sedm rozhraní sloužící k práci s jazykovými elementy. Jsou to IOperand a jeho potomci IUdaj, IOdkaz, IFunkce, IVolaniFunkce a rozhraní IOperace. Dále jsou zde dvě rozhraní pro práci s příkazy kódy IPrikaz a IKodProgramu.



Obrázek 1: Třídní diagram řízení kódu a generování událostí



Obrázek 2: Sekvenční diagram rozesílání událostí

název události	**příležitost**	**parametry**
programNacten	načtení kódu algoritmu do stroje	načtený kód
programUkoncen	ukončení provádění kódu	*bez parametrů*
pripravPrikaz	před provedením příkazu	příkaz
pripravProveden	provedení příkazu	příkaz
deklaracePromenneProvedena	provedení deklarace proměnné	datový typ, název proměnné a inic. výraz
prikazVyrazProveded	provedení vyhodnocení výrazu	vyhodnocený výraz
bunkaVPametiZrusena	vymazání buňky z paměti	název buňky
bunkaVPametiVytvorena	vytvoření buňky v paměti	datový typ, název buňky, vnoření paměti
doBunkyUlozenUdaj	uložení údaje do buňky	název buňky, uložený údaj, vnoření paměti
zBunkyPrectenUdaj	čtení údaje z buňky	název buňky, vnoření
operaceProvedena	provedení operace	oba operandy a operace
doPromenneUlozeno	uložení údaje do proměnné	název proměnné, uložený údaj
doPromenneUlozenoZOdkazu	přenesení údaje z odkazu do proměnné	název proměnné, zdrojový odkaz
doPromenneVPoliUlozeno	uložení údaje do pole	název pole, uložený údaj, indexy
doPromenneVPoliUlozenoZOdkazu	přenesení údaje z odkazu do pole	název pole, zdrojový odkaz, indexy
poleInicializovano	inicializace pole	název pole, rozměry pole
funkceZavolana	zavolání triviální funkce	název funkce, výrazy předané jako parametry
uzivatelskaUdalost	uživatelem definovaná událost	obecný java.Object

Tabulka 1: Události generované aplikačním rámcem

4.2.1 IOperand

Rozhraní slouží k zaobalení dat. Hodnotu operandu lze získat metodou vyhodnotSe(**int**). Tato metoda má jeden parametr typu **int**, který určuje zda-li se při vyhodnocení smí zapisovat do paměti. Konstanty k tomuto parametru jsou definovány rovněž v tomto rozhraní a mohou nabývat hodnot JEN_PRO_CTENI a CTENI_A_ZAPIS. Toto rozhraní je dále rozšířeno.

4.2.2 IUdaj

Rozhraní rozšiřující IOperand. Rozhraní reprezentuje primitivní datové typy. Obsahuje metodou dejTypUdaje(), vracející název datového typu. Metoda jeRovno(IUdaj) slouží k porovnání dvou údajů. Rozhraní dědí ze standardních rozhraní Cloneable a Comparable

4.2.3 IOdkaz

Další rozhraní rozšiřující IOperand. Reprezentuje všechny druhy proměnných (klasické proměnné, proměnné v poli, délku pole apod.). Toto rozhraní lze také použít k definici veřejných polí v případných třídách. Pro čtení hodnoty lze použít zděděnou metodu vyhodnotSe(**int**). Pro uložení je k dispozici dvojice metod ulozUdaj(IUdaj) pro přímé vložení údaje a ulozUdajZOdkazu(IOdkaz, **int**) pro přesun údaje z odkazu do odkazu; parametr **int** potom určuje zda-li se smí při přesunu zapisovat do paměti. Rozhraní dále definuje metodu dejNazevBunky() vracející název buňky použitý v paměti, dejTypUdaje() vrací název typu údaje uloženého v odkazu a existujeBunka() určuje zda-li odkaz odkazuje na existující buňku v paměti.

4.2.4 IFunkce

Rozhraní IFunkce reprezentuje triviální funkce. Touto funkcí je např. random, nahodnePole ale také funkce reprezentující konstruktory novePole, novyString. Kromě klasické metody vyhodnotSe(), které slouží k vyhodnocení funkce obsahuje metody dejNazev() vracející název funkce, metodu dejParametry() vracející pole parametrů, kde každý parametr je datového typu Vyraz. Poslední metoda jeTvorici () vrací logickou hodnotu určující zda-li se při prezentování funkce v kódu má před název vložit klíčové slovo **new**.

4.2.5 IVolaniFunkce

Toto rozhraní má pouze jednu funkci a to rozlišit klasické operandy od volání netriviální, tedy uživatelem definované funkce. Výrazy obsahující volání těchto funkcí nejsou přímo vyhodnotitelná viz. 4.3.1.

4.2.6 IOperace

Rozhraní definuje metody pro práci s operacemi (např. sčítání, inkrementace, přiřazení apod.). Tři metody dejPrioritu () , jeBinarni () , jeAsociativniZPrava() slouží k zjištění vlastnosti

operací. K vykonání operace potom lze použít metodu provedOperaci(IOperand, IOperand), tato metoda má dva parametry typu IOperand a jeden parametr typu **int**. Výsledek je vrácen ve formě IOperand.

4.2.7 IPrikaz

Rozhraní reprezentuje jednotkový příkaz. Obsahuje metodu proved() sloužící ke spuštění příkazu. Dále jsou v rozhraní definovaný metody nastavRodicovskyKod(IPrikaz) a dejRodicovskyKod(), tyto metody slouží k nastavení a vrácení rodičovského kódu. Struktura je popsaná v kapitole 5. K navrácení prohlížeče kódu slouží metoda dejProhlizecKodu.

4.2.8 IKodProgramu

Rozhraní reprezentuje kompletní kód prezentovaného algoritmu. Stejně jako příkaz obsahuje metody proved() a dejProhlizec(). Navíc předepisuje metody pro práci se skoky jsou to skocNaFunkci(String) a skocNaAdresy(List<Integer>). Parametry spuštění kódu lze měnit metodou nastavParametry(Vyraz...), opětovně je získat lze metodami dejTypyParametru() dejNazvyParametru().

4.3 Návrh a implementace řešení výrazů

Vyhodnocování výrazu je jednou z nejdůležitějších součástí vykonávání kódu. Výraz je složen z dat a operací mezi nimi. Zatímco například kalkulátor si vystačí z čísly a jednoduchými operacemi u vykonávání kódu jsou potřeba i symbolické odkazy jako proměnné nebo volání funkcí.

Můj návrh vychází z myšlenky použití binárního stromu [3]. Binární strom je nelineární datová struktura obsahující uzly, které mohou odkazovat na další dva uzly, potomky. Každý uzel může mít tedy nanejvýš dva potomky a každý uzel má nanejvýš jednoho rodiče. Uzel, který nemá rodiče,to je ten na vrcholu stromu, se nazývá kořen. Uzly, které nemají potomky, ty v nejspodnějším patře stromu, se nazývají listy. Každý binární strom může obsahovat několik listů, ale kořen obsahuje vždy pouze jeden.

Struktura binárního stromu je pro vyhodnocování výrazu výhodná. Jednotlivé uzly uvnitř stromu reprezentují operace a koncové uzly, listy, naopak reprezentují data. Chceme-li aritmetický výraz vyhodnotit, vyhodnotíme kořenový uzel stromu. Jsou-li v tomto kořenovém uzlu data pak jsou tyto data výsledkem výrazu. Je-li v kořenovém uzlu operace, vyhodnotí se nejdříve levý a pravý potomek a následně se nad výsledky spustí operace. Výsledek této operace je pak výsledek celého výrazu. Potomkem operace samozřejmě může být další operace, uzly se tedy vyhodnocují rekurzivně od kořene k listům.

Skutečností, která tento způsob mírně komplikuje, je různá priorita různých operací, např. násobení má přednost před sčítáním apod.. V programovacím jazyce Java je těchto úrovní priorit celkem 13. Při skládání stromu reprezentujícího výraz musím tedy dodržet to, aby se operace s vyšší prioritou nacházeli ve spodních patrech stromu blíže datům a operace s nižší prioritou naopak ve vyšších patrech blíže kořenu. Protože se strom

vyhodnocuje rekurzivně nejdříve se budou spouštět operace ve spodních patrech, tedy ty s vyšší prioritou.

Druhým aspektem ovlivňujícím skládání stromu je asociativita operací. Asociativita definuje pořadí operací se stejnou prioritou. Operace mohou mít asociativitu levou nebo pravou. Levou asociativitu má například operace odčítání, napíšeme-li výraz $4 - 3 - 2$, vyhodnocujeme jej jako $(4 - 3) - 2$, výraz je uzávorkovaný na levé straně, proto levá asociativita. Pravou asociativitu má například operace přiřazení, a = b = c se vyhodnotí jako a = (b = c), na rozdíl od předchozího příkladu nyní je výraz uzávorkován na pravé straně. V mém případě budu tento jev řešit tak, že operace s pravou asociativitou budou mít přednost před operacemi s levou asociativitou. Tato přednost se bude samozřejmě uplatňovat pouze tehdy, budou-li mít operace stejnou prioritu.

V programovacím jazyce Java se kromě klasických binárních operací vyskytují i operace unární a ternární. Unární operace pracují pouze s jedním operandem (typická operace inkrementace x++). Ternární operace má naproti tomu operandy tři (operace x ? y : z). Zakomponovat unární operace do binárního stromu je jednoduché, místo dvou potomku bude mít unární operace potomka pouze jednoho. Bude-li operátor stát za operandem bude se jednat o potomka pravého, bude-li stát před operandem bude se pak jednat o potomka levého. Dle zadání ternární operace implementovat nebudu. V příloze C jsou popsány všechny operace jazyku Java.

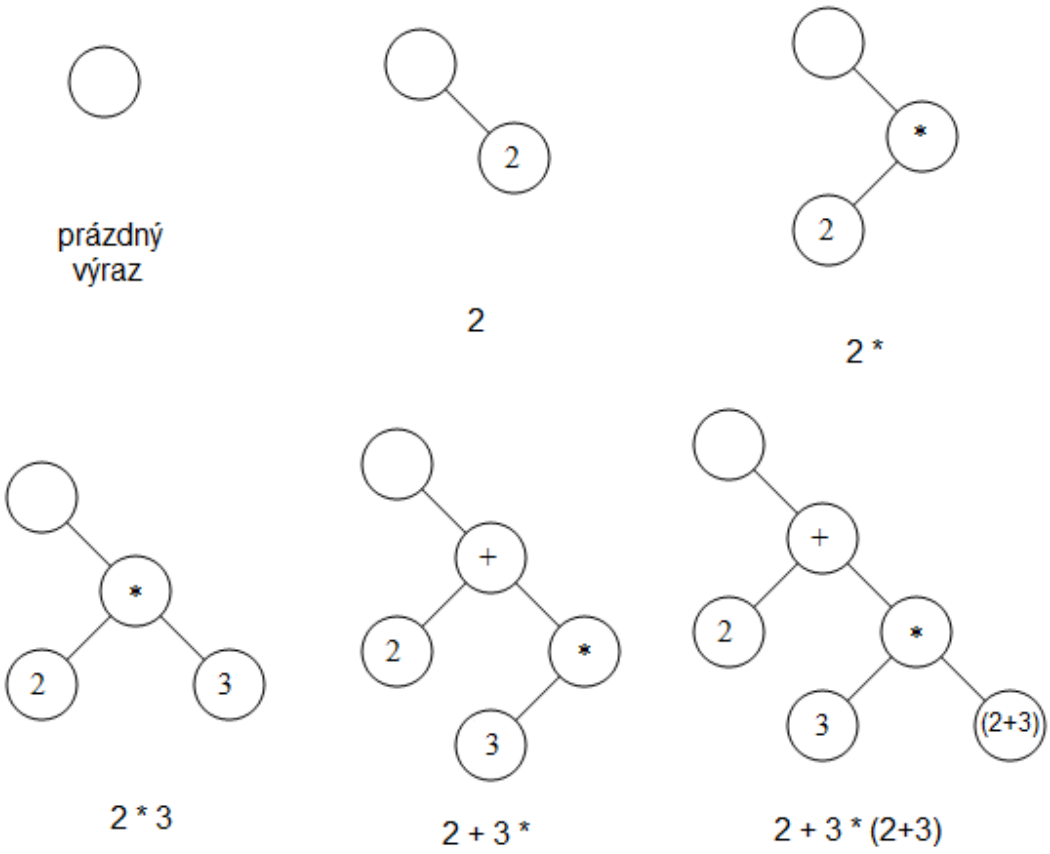
4.3.1 Návrh implementace binárního stromu

Výrazy budou reprezentovány binárním stromem, který se skládá z uzlů. Uzly obsahují informace (operace nebo data). Jelikož objekty reprezentující operace a data budeme potřebovat i v jiných částech aplikačního rámce, je vhodné je oddělit od samotné implementace binárního stromu.

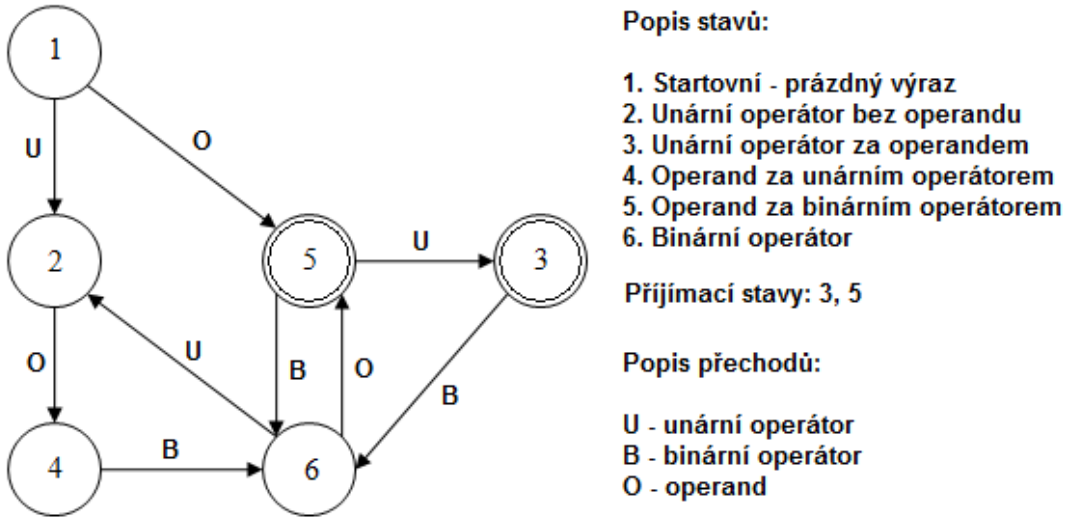
Základem implementace binárního stromu bude objekt reprezentující uzel stromu. Uzly obsahují informaci o tom co uzel obsahuje, touto informací může být operace, operand případě jiný výraz. Každý uzel bude dále charakterizován váhou, váha bude určovat postavení uzlu ve stromě. Nejvyšší váhu budou mít operandy a jiné výrazy, u uzlu s operací bude váha určená prioritou a asociativitou operace. Výpočet váhy uzlu s operací: VAHA = 2*PRIORITA, v případě že operace bude asociativní zprava, přičteme k této váze 1. Tím zajistíme aby uzly s operacemi s vyšší prioritou byli těžší. Protože uzly s operandy musí být vždycky v nejspodnějším patře stromu přiřadíme jim váhu MAX_INTEGER. Uzly s vyšší vahou budou klesat dolů do spodních pater, tím zajistíme správné pořadí vyhodnocování operací.

Způsob napojení uzlů do stromu závisí na váze uzlu. Na začátku sestavování je prázdný výraz, tento výraz obsahuje jeden uzel, který slouží k pozdějšímu napojení dalších uzlů a nenese žádnou užitečnou informaci. Při napojení dalších uzlů se uplatní algoritmus zobrazený ve výpise 1.

Sestavení výrazu $2 + 3 * (2+3)$ demonstruje obrázek 3. Operace sčítání má nižší prioritu než operace násobení, proto je ve stromě výše. Uzel s výrazem (2+3) obsahuje vlastní binární strom. Tímto způsobem budu implementovat závorky.



Obrázek 3: Ukázka sestavení binárního stromu



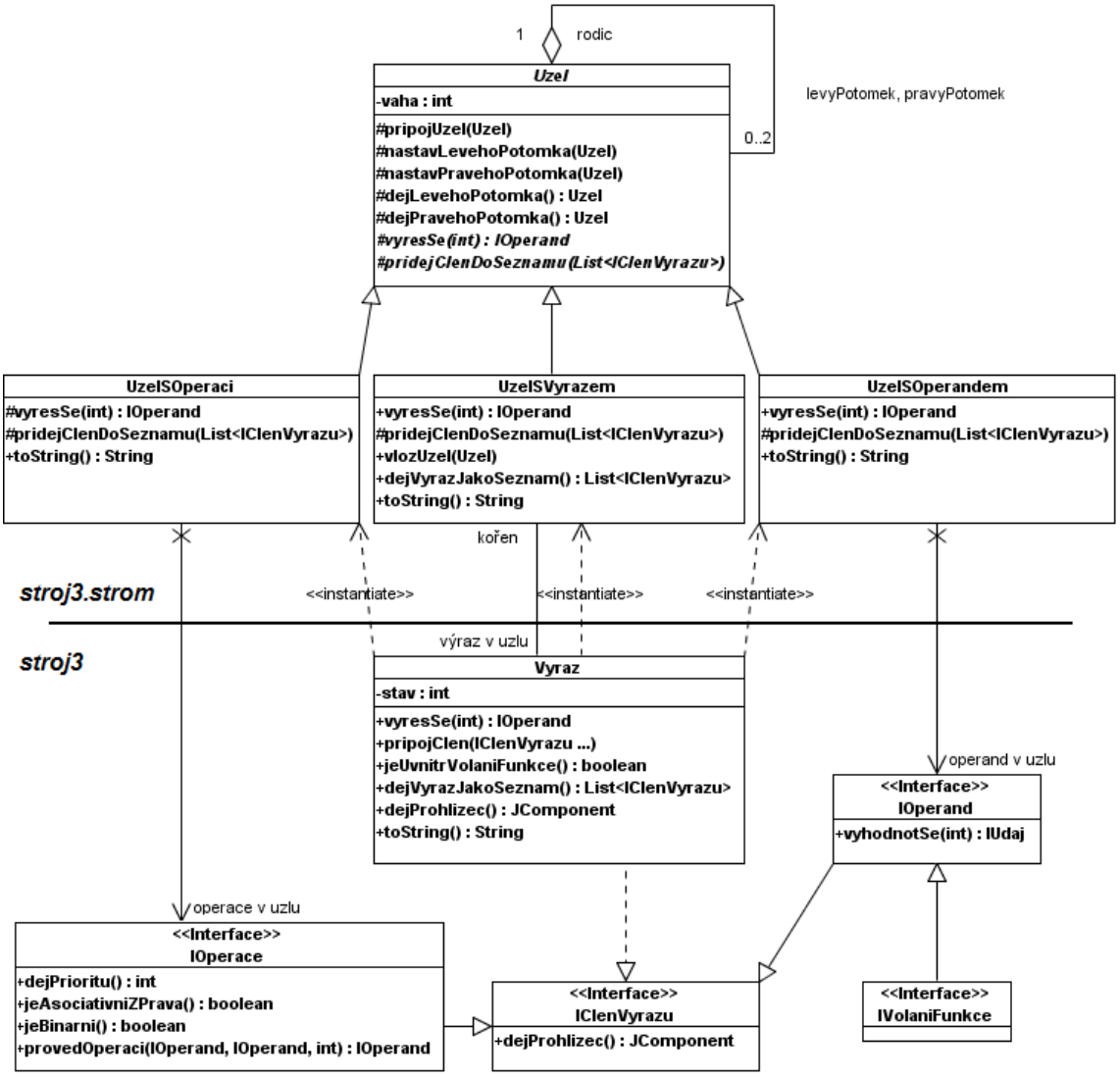
Obrázek 4: Automat přijímací výrazy

Dalším problémem, který je nutno vyřešit je zajistit, aby výraz byl syntakticky správně sestaven. Například není povoleno zapsat $2 + * 5$ nebo $++a--$. Tuto kontrolu budu provádět hned při vkládání členů výrazu do výrazu. Členy výrazu si rozdělíme na operandy, binární operace a unární operace. S každým příchozím členem se výraz překlopí do příslušného stavu, nemůže-li se překlopit, protože příslušný přechod z příslušného stavu neexistuje, sestavování výrazu skončí vyjímkou. Pokusím-li se vyhodnotit výraz ve stavu, kdy není ukončen, skončí pokus opět vyjímkou. Obrázek 4 zobrazuje příjimací automat.

Ve výrazech se mohou objevit také proměnné či volání funkcí. Proměnné se budou interpretovat jako operandy. Volání funkcí se budou dělit na dva druhy. Prvním budou jednoduché přímo implementované vestavěné funkce (př. náhodné číslo, sinus ...). Tyto funkce nebudou nějak rozepisovány a budou mít tedy stejně jako proměnné status operandu. Druhým druhem pak bude volání rozepsaných funkcí, tyto volání nepůjde vyhodnotit přímo, ale bude se muset počkat na to až proběhne příslušný kód. Z toho vyplývá, že výraz jež bude obsahovat jedině volání rozepsané funkce nebude přímo vyhodnotitelný, ale bude se muset rozepsat do více kroků.

4.3.2 Implementace řešení výrazů

Implementace řešení je znázorněna na obrázku 5. Základem je abstraktní třída Uzel, která drží odkazy na tři další uzly, levého potomka, pravého potomka a rodiče a navíc obsahuje atribut váha. Ze třídy uzel dědí třídy UzelSOperaci, UzelSOperandem, UzelSVyrazem). Každá z těchto tří tříd obsahuje příslušný objekt. UzelSOperaci objekt implementující rozhraní IOperace, které definuje předpis pro práci s operacemi, UzelSOperandem objekt implementující rozhraní IOperand, které definuje předpis pro práci s operandy a nakonec UzelSVyrazem obsahující samotný výraz, tato asociace je na rozdíl od ostatních obou-



Obrázek 5: Třídní diagram řešení výrazů

směrná. Poslední rozhraní IClenVyrazu má dvě funkce, za prvé zaobaluje všechny objekty jež mohou vystupovat jako členové výrazu a za druhé předepisuje metodu dejProhlizec() vracející JComponent, která je následně použita k prezentaci člena výrazu.

Sestavení výrazu se provede vytvořením nové instance třídy Vyraz a následným vkládáním členů metodou pripojClen(). Metoda pripojClen() má několik přetížení vždy je ale parametrem této metody jeden či více členů výrazu. Na základě druhu člena výrazu se automaticky na vstupu metody pripojClen() mění vnitřní atribut stav, tím se kontroluje správná syntaxe výrazu (viz. obrázek 4). Pokud tedy člen výrazu projde touto kontrolou, vytvoří se instance příslušného uzlu (UzelSOperaci, UzelSOperandem nebo UzelSVyrazem, který se následně ke stromu připojí ke kořeni metodou vlozUzel(). Tato metoda vnitřně volá metodu pripojUzel nalézající se ve třídě Uzel. V metodě pripojClen je implementován algoritmus 1.

Vyřešení výrazu se provede zavoláním metody vyresSe(**int**). Metoda má parametr, který určuje zda-li se smí při řešení zapisovat do paměti (díky přiřazovacím operacím by jinak nešlo vyřešit výraz „nanečisto“). V metodě se zkontroluje stav výrazu a to zda-li neobsahuje volání netriviální funkce (otestovat zda-li se ve výrazu nalézá netriviální funkce lze metodou jeUvnitrVolaniFunkce()). Projde-li výraz kontrolou, zavolá se na kořen metoda vyresSe(). Každá z trojice tříd UzelSOperaci, UzelSOperandem, UzelSVyrazem tuto metodu implementuje jinak. UzelSVyrazem zkontroluje zda-li není prázdný, pokud ne zavolá metodu vyresSe() na svůj startovní uzel. Třída UzelSOperandem při řešení vrátí přímo operand v něm obsažený. Třída UzelSOperaci nejprve vyřeší levého a pravého potomka a výsledky pak předá jako parametry metodě provedOperaci na rozhraní IOperace . Řešením výrazu je vždy objekt implementující rozhraní IOperand.

```
protected void pripojUzel(Uzel u) {

    boolean vahaJeVyssi = (u.vaha > vaha);
    boolean vahaJeRovna = (u.vaha == vaha);
    boolean vahaJeLicha = (u.vaha % 2) == 1;
    if (vahaJeVyssi || (vahaJeRovna && vahaJeLicha)) {

        if (pravyPotomek == null)
            nastavPravehoPotomka(u);
        else
            pravyPotomek.pripojUzel(u);

    } else {

        rodic.nastavPravehoPotomka(u);
        u.nastavLevehoPotomka(this);

    }

}
```

Výpis 1: Způsob napojení uzlů do stromu

Zobrazení výrazu, protože je výraz zapsán ve stromové struktuře, je třeba nejprve vytvořit seznam členů, tak jak byli do výrazu vloženy za sebou. K tomu slouží metoda

dejVyrazJakoSeznam(). Poté co máme výraz ve formě jednoduchého seznamu. Můžeme na každého člena zavolat metodu dejProhlizec() definovanou rozhraním IClenVyrazu. Každý člen výrazu, pak metodu dejProhlizec() tak, že volá továrnu na prohlížeče, která je součástí prezentační části.

4.4 Návrh a implementace paměti

Každý mechanismus vykonávající nějaký programový kód potřebuje ke své činnosti paměť. V této paměti se ukládají hodnoty proměnných nebo hodnoty návratových adres při volání funkcí.

Každá proměnná je reprezentována svým názvem, který slouží jako symbolicky odkaz na paměťovou buňku. V této buňce může být hodnota uložena buď přímo nebo nepřímo jako adresa odkazující na jinou buňku. Java používá k uložení proměnných oba způsoby. Prvním přímým způsobem se ukládají primitivní datové typy, druhým nepřímým způsobem potom typy referenční.

4.4.1 Návrh paměti

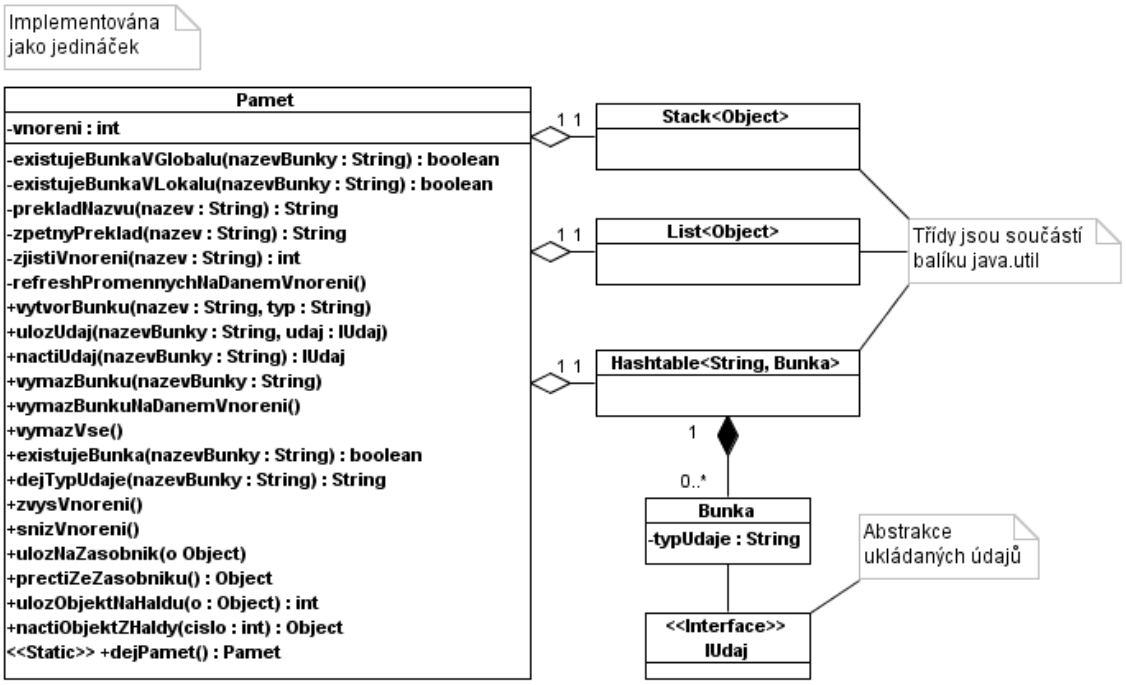
Základem implementace paměti bude tedy hašovací tabulka, která ukládá dvojice klíč – hodnota. Klíčem v tomto případě bude textový řetězec reprezentující název proměnné a hodnotou pak vlastní hodnota této proměnné. Klíče, tedy názvy proměnných samozřejmě musí být unikátní, avšak může se stát, že proměnné stejného názvu se budou vyskytovat v několika různých funkcích. Bude tedy nutné názvy takových proměnných rozlišit. V paměti se tedy bude udržovat hodnota vnoření. S každým voláním funkcí se tato hodnota zvýší o jedničku a každým návratem z funkce se potom o jedničku sníží. Bude-li chtít krokovaný algoritmus vytvořit proměnnou automaticky se vloží před její název číslo tohoto vnoření. Proměnné, které nebudou vytvářený ve funkcích, ale budou globálně k dispozici pro všechny funkce nebudou toto číslování používat. Bude-li chtít krokovaný algoritmus s proměnné číst nebo do ní zapisovat, prohledá se nejprve dané vnoření (lokálně vytvořené proměnné) a až poté proměnné bez vnoření (globálně vytvořené proměnné). Tímto způsobem se zbavím i problému s vytvářením proměnných v rekurzivně volaných funkcích.

Referenční proměnné se budou ukládat stejným způsobem jako proměnné primitivní s tím rozdílem, že hodnota v hašovací tabulce bude obsahovat číslo. Toto číslo bude reprezentovat index v jednoduchém seznamů objektů, který bude také součástí paměti. Je tedy umožněno, aby více proměnných ukazovalo na stejný objekt.

Poslední důležitou součásti paměti je zásobník. Do zásobníku se budou ukládat návratové adresy během volání netriviálních funkcí a také jejich návratové hodnoty. Celý mechanismus podrobně popíšu později.

4.4.2 Implementace paměti

Třídní diagram je znázorněn na obrázku 6. Základem je třída Pamet. Protože není nutné aby bylo v systému více instancí paměti je tato třída implementována jako jedináček (podle



Obrázek 6: Třídní diagram paměti

návrhového vzoru Singleton). Instanci paměti pak lze získat statickou metodou dejPamet (). Paměť obsahuje tří základní kolekce, Hashtable na ukládání dvojic název proměnné – hodnota, Stack reprezentující zásobník a List reprezentující haldu. Všechny tyto kolekce jsou součástí standardního balíku *java.util*[2].

Klíčem v kolekci Hashtable je textový řetězec String a hodnotou potom instance třídy Bunka, jež je vnitřní třídou třídy Pamet a v podstatě reprezentuje proměnnou. Třída buňka obsahuje textový řetězec s názvem datového typu a odkaz na rozhraní IUdaj. Rozhraní IUdaj představuje abstrakci všech primitivních datových typů používaných v aplikačním rámci. Důležité je, že pří ukládání a čtení údajů se předávají vždy kopie těchto objektů.

Vytváření buněk Buňku lze vytvořit metodou vytvorBunku(String, String), metoda má dva parametry, první určuje název buňky a druhý název datového typu. Před název buňky se automaticky vloží číslo vnoření oddělené ':'. Formát názvu je vnoreni:nazev. Vyjímku tvoří globálně vytvořené buňky, které nemají před názvem číslo s vnořením. Metoda vytvorBunku(String, String) se volá vždy při deklaraci proměnné. Protože název buňky musí být unikátní je nutné při vytváření buněk kontrolovat, zda-li buňka s daným názvem v paměti již neexistuje. Této kontroly se ale účastní jen buňky na aktuálním vnoření.

Číslo vnoření je soukromý atribut třídy Pamet. Po vytvoření nebo vymazání paměti je hodnota vnoření automaticky nastavena na hodnotu −1, což znamená že se paměť nachází v globálním prostoru. Hodnotu vnoření lze měnit metodami zvysVnoreni() a snizVnoreni().

název události	příležitost
bunkaVPametiVytvorena	vytvoření nové buňky
bunkaVPametiZrusena	vymazání buňky v paměti, snížení vnoření
zBunkyPrectenUdaj	čtení údaje z buňky
doBunkyUlozenUdaj	uložení údaje do buňky

Tabulka 2: Události generované v paměti

Ukládání a čtení hodnoty lze provést metodou ulozUdaj(String, IUdaj). Prvním parametrem je název buňky (pouze čistý název, hodnotu vnoření si pamět' doplní sama) a druhým potom údaj, jež se do buňky ukládá. Aby se uložení zdařilo, je nutné splnit dvě podmínky. První podmínkou je existence buňky v paměti. Při zjišťování existence buňky se nejprve prohledá aktuální vnoření a poté globální prostor, tedy buňky jež nemají před názvem číslo. V případě nenalezení dané buňky je vyvolaná vyjímka. Druhou podmínkou je potom shoda datových typů. Při vyhledání buňky se přečte název jejího datového typu a ten se porovnává s názvem datového typu údaje předaného v parametru. Analogicky probíhá čtení hodnoty údaje. Rozdílem je, že volá metoda nactiUdaj(String) a odpadá také kontrola datového typu.

Zásobník uchovává obecné objekty. K práci ze zásobníkem slouží dvojice metod ulozNaZasobnik(), prectiZeZasobniku(). Tyto metody v podstatě jen delegují metody push() a pop(), které jsou součástí Javovské třídy Stack[2].

Halda uchovává obecné objekty. Metoda ulozObjekNaHaldu(Object) uloží objekt na haldu a vrátí jeho index, pomocí nějž můžeme později k objektu znovu přistoupit. Metoda nactiObjekZHaldy() potom přečte z haldy objekt uložený na daném indexu. Index se ukládá do třídy UdajSPtr, která v aplikačním rámci reprezentuje referenci.

K mazaní paměti jsou k dispozici tři metody. Metoda vymazBunku() vymaže konkrétní buňku z hašovací tabulky. Metoda vymazBunkyNaDanemVnoreni() pak vymaže z hašovací tabulky všechny buňky nacházející se na daném vnoření. Tato metoda se volá těsně před návratem z funkce a to proto, aby se pamět'očistila od lokálních proměnných v této funkci deklarovaných. Poslední mazací metoda vymazVse(), slouží k vymazaní celé paměti, tzn. všech buňek, zásobníku i haldy, metoda navíc nastaví hodnotu vnoření na -1. Metoda se volá při restartu algoritmu nebo načtení algoritmu nového.

Generování událostí. Pamět'jako i ostatní části aplikačního rámce generují události na které pak reaguje prezentační část. Přehled událostí generovaných pamětí je zobrazen v tabulce 2. Princip generování, odposlouchávání událostí včetně jejích podrobného popisu je vysvětlen v podkapitole 4.1.

Soukromé pomocné metody

Mimo základních veřejných metod obsahuje třída Pamet i sadu pomocných soukromých metod.

Metody na převod názvu buněk. Metoda prekladNazvu(), přidá před název buňky číslo aktuálního vnoření. Metoda zpetnyPreklad() naopak z názvu buňky číslo vnoření odebere. Metodou zjistiVnoreni () lze zjistit vnoření, při kterém byla buňka vytvořena. Metodami existujeBunkaVLokalu() a existujeBunkaVGlobalu() se testují existence buňky v

lokálním respektive globálním prostoru. Poslední metoda refreshBunekNaDanemVnoreni() se automaticky volá při snížení vnoření.

5 Návrh a implementace příkazů a kódu

Příkazy a kódy jsou základními kameny skládání algoritmů. Představují jednoduché příkazy jako deklarace proměnné, vyřešení výrazu apod. ale také řeší složené bloky kódu, větvení, cykly, volání funkcí apod. Všechny tyto entity jsou implementovány v balíku kody. Postup jak sestavovat algoritmus pomocí níže popsaných tříd je v příloze A.

5.1 Struktura a dělení

Mezi základními jazykovými prvky popsanými v kapitole 4.2 je i rozhraní IPrikaz, které zastřešuje jednotkový příkaz. Slovem příkaz je chápán jak jednoduchý úkon (vyhodnocení výrazu, deklarace proměnné, skok atd.), tak celý blok příkazů (if-else, cykly, funkce atd.). Abych tyto dva chápání rozlišil, budu jednoduchý příkaz nazývat *příkaz* a zatímco bloku příkazů budu říkat *kód*.

Každý kód obsahuje lineární seznam objektu implementující rozhraní IPrikaz. Příkaz i kód toto rozhraní implementují, kód se tedy může skládat nejen z příkazů ale i z jiných kódu, které obsahují další příkazy a kódy, které mohou obsahovat další příkazy a kódy atd. Vzniká tak v podstatě stromová struktura [4]. Mimo seznam příkazů obsahuje kód i seznam proměnných, které byly deklarovány příkazy jež obsahuje. Po ukončení kódu se podle tohoto seznamu provede vymazání příslušných proměnných z paměti.

Celý vizualizovaný algoritmus je pak reprezentován seznamem kódu (jednotlivými funkcemi) a seznamem příkazů (globální deklarace proměnných).

Struktura je znázorněna na obrázku 7.

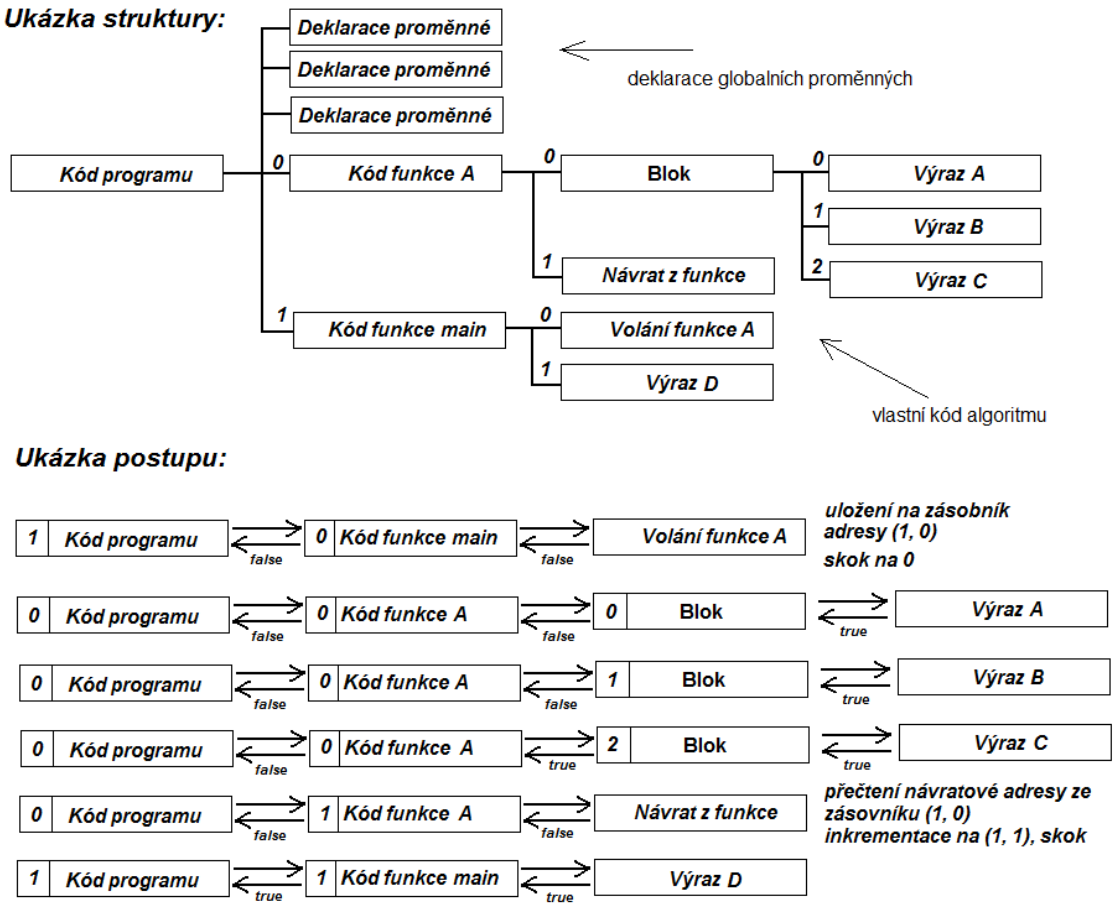
5.2 Způsob procházení kódu

Příkazy obsažené v kódu jsou seskupeny do jednoduchého seznamu, přistupovat k nim lze tedy pomocí indexu. Je-li na kód zavolána metoda proved(), kód tuto metodu přepošle příkazu na který zrovna ukazuje index. Po provedení každého příkazu je vrácena logická hodnota, která říká zda-li má nadřazený kód inkrementovat svůj index ukazující na aktuální příkaz. Pokud index ukazuje mimo, znamená to že všechny příkazy v kódu byly provedeny a sám kód vrátí svému nadřazenému kódu **true**, tedy dá mu najevo že musí zase svůj index posunout.

V případě větvení nebo cyklů je tato linearita narušována, kódy proto mohou přetěžovat metodu inkrementujAdresuPrikazu() a řídit tak tok kódu (cyklus například po skončení se automaticky vrací na začátek) nebo mohou obsahovat příkazy skoku (většinou podmíněného), které mají právo měnit index svého nadřazeného kódu a tak řídit pořadí prováděných příkazů.

5.3 Volání funkcí a návrat zpět

Lineární procházení kódu či skoky jsou vždy prováděny v rámci jednoho kódu. V případě volání funkcí však nelze tento postup využít. Funkce nejsou do sebe zanořovány, ale jsou v podstatě sourozenci kódem algoritmu. Kódy funkcí do sebe zanořovat nelze, protože

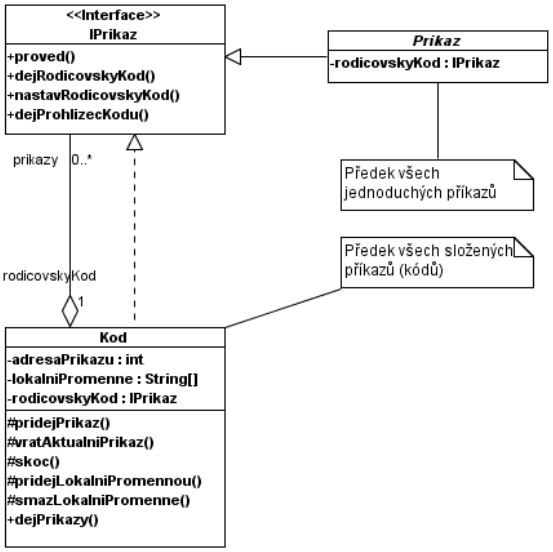


Obrázek 7: Ukázka struktury kódu a jeho procházení

v případě rekurzivního volání by docházelo k zacyklení. Volání funkcí jsou tedy skoky napříč stromovou strukturou. Stejným problémem jsou skoky zpět, tedy návraty z takto volaných funkcí.

Jak je uvedeno výše, struktura kódu je stromovitá. Kořenem tohoto stromu je kód programu, ten obsahuje odkazy na další kódy (jednotlivé funkce), které jsou složený z dalších kódů nebo příkazů. Každý příkaz lze tedy snadno adresovat pomocí seznamu indexů jednotlivých kódů. Má-li se tedy provést zavolání funkce zjistí se adresa tohoto volání (adresa je reprezentována seznam integerů) uloží se na zásobník do paměti (tím se zapamatuje místo návratu). Druhým krokem je zjištění cíle, tedy adresy volané funkce. A nakonec se na kořen zavolá se metoda skoč na adresu. Tím se řízení předá dané funkci (v podstatě se změní index kódu algoritmu, případně jeho potomků). V případě návratu z volané funkce je postup obdobný, adresa návratového místa se získá vyzvednutím ze zásobníku v paměti. Tímto způsobem lze zajistit i rekurzivní volání funkcí.

Celý postup je znázorněn na obrázku 7.



Obrázek 8: Třídní diagram kódů a příkazů

5.4 Implementace

Na obrázku 8 je znázorněn třídní diagram struktury kódu a příkazů; jedná se o návrhový vzor Kompozit. Příkaz je reprezentován abstraktní třídou Prikaz. Jedinou její funkcí je, že drží odkaz na svůj rodičovský kód a umožňuje tak k němu přistupovat. Ostatní metody zůstávají abstraktní a je na jednotlivých příkazech aby si je implementovaly. Kódy jsou reprezentovány třídou Kod tato třída drží stejně jako příkazy odkaz na svůj rodičovský kód. Dále drží seznam názvů lokálních proměnných a seznam obsažených příkazů.

5.4.1 Příkazy

DeklaracePromenne je reprezentována třídou PrikazDeklaracePromenne. Deklarace proměnné je definována názvem proměnné, názvem typu proměnné a inicializačním výrazem. Po zavolání metody proved() se v paměti vytvoří nová proměnná daného názvu a typu, název proměnné se přidá do seznamu lokálních proměnných nadřazeného kódu a nakonec se provede vyhodnocení inicializačního výrazu. Metoda proved() vždy vrací **true**. (V případě, že je v inicializačním výrazu volání používá se KodDeklaracePromenne)

Vyraz je reprezentován třídou PrikazVyraz. Obsahuje pouze jeden výraz, který se provede po zavolání metody proved(), metoda stejně jako v předešlém případě vrací vždy **true**. (V případě, že je v inicializačním výrazu volání používá se KodDeklaracePromenne)

Skok je reprezentován třídou PrikazPodminenySkok. Skok je definován podmínkou za které se má skok provést a adresu kam se má skočit (adresa je v rámci jednoho nadřazeného kódu). Po zavolání metody proved() se provede vyhodnocení podmínky, je-li splněna, je nadřazenému kódu přepsán index ukazující na aktuální příkaz (tím se provede skok), metoda poté vrátí **false**, aby se nadřazený kód automaticky neposunul.

V případě, že podmínka splněna není metoda vrací **true**, tím se provede standardní posun na další příkaz. Adresa skoku se dá měnit metodami nastavAdresuSkoku() nebo InkrementujAdresuSkoku(). Tento příkaz nestojí nikdy samostatně a je vždy součástí kódu větvení nebo cyklu.

Volání funkce je reprezentováno třídou PrikazVolaniFunkce. Volání funkce je definováno pouze názvem volané funkce. Po zavolání metody proved() se provede vyhledání a skok na danou funkci. Metody vždy vrací **false**.

Hlavička funkce reprezentována třídou PrikazHlavickaFunkce se nachází výlučně na začátku každé funkce. Po zavolání metody proved() deklaruje parametry funkce. V rámci této deklarace se provede deklarace jednotlivých proměnných reprezentující parametry a uloží do nich předané hodnoty. Metoda proved() v této třídě vždy vrací **true**.

NavratZFunkce je reprezentován třídou PrikazNavratZFunkce. Návrat z funkce může obsahovat výraz. Po zavolání metody proved(), se ze zásobníku v paměti přečte návratová adresa, do zásobníku zpět se uloží případná vrácena hodnota. Provede se skok na návratovou adresu. Metoda vrací **false**, jedině v případě že se jedná o poslední příkaz ve funkci main je vráceno **true**.

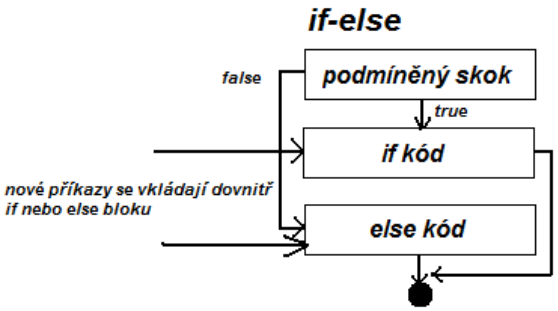
5.4.2 Kódy

KodVyraz,**KodNavratZFunkce**,**KodDeklaracePromenne** Kód výraz je speciálním případem příkazu s výrazem. Používá se když je ve výrazu volání funkce. Výraz z voláním funkce není přímo vyhodnotitelný proto se musí rozepsat do více kroků. Tento kód je reprezentován třídou KodVyraz. Výraz se rekurzivně projde, naleznou se všechna volání funkcí, které se postupně volají. Po každém volání funkce se aktualizuje hodnota výrazu (volání funkcí se nahrazují výsledky těchto volání). Na úplný konec se provede standardní příkaz vyhodnocení výrazu. V případě KodNavratZFunkce nebo KodDeklaracePromenne je postup stejný až na to že na konci kódu se navíc provede příkaz návrat z funkce resp. deklarace proměnné.

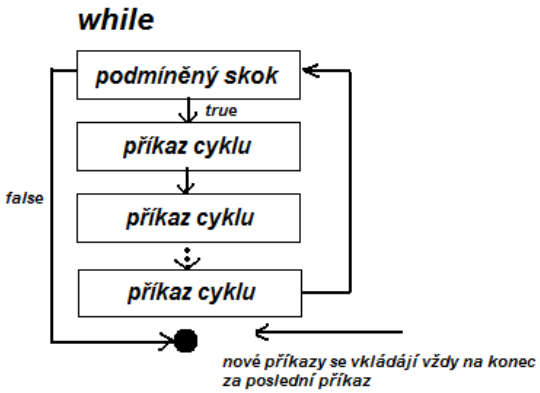
KodIfElse, obrázek 9 Blok **if**−**else** reprezentovaný třídou KodIfElse se skládá ze dvou Kod a jednoho podmíněného skoku PrikazPodminenySkok. V případě splněny podmínky se provede blok kódu if, else se po vykonání přeskočí, pokud podmínka splněna nebude řízení se předá přímo bloku else.

KodWhile, obrázek 10 Cyklus s podmínkou na začátku tedy **while**, je reprezentován třídou KodWhile. Při vytvoření tohoto kódu se automaticky vytvoří na začátku příkaz podmíněný skok, který ukazuje na konec celého cyklu. Není-li podmínka splněna řízení je přesměrováno na konec, v opačném případě se kód prochází lineárně do té doby než dojde na poslední příkaz, po té je automaticky přesměrován zpět na počáteční podmínku. Při vkládání nových příkazů do cyklu se automaticky zvyšuje skok v počáteční podmínce.

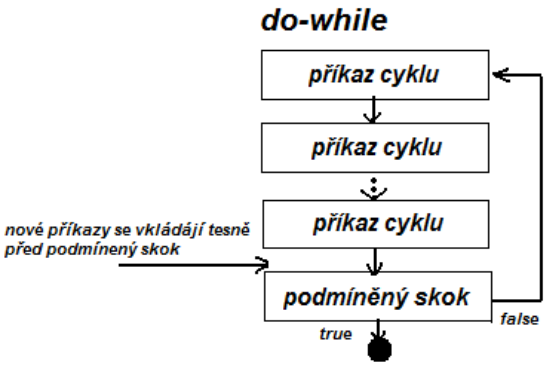
KodDoWhile, obrázek 11 Cyklus s podmínkou na konci tedy **do**−**while**, je reprezentován třídou KodDoWhile. Funguje na stejném principu jako předchozí cyklus, s tím rozdílem, že podmíněný skok je na konci a ukazuje směrem na začátek. V případě vkládání nových příkazů do cyklu se podmíněný skok automaticky posouvá, takže zůstává vždy jako poslední příkaz bloku.



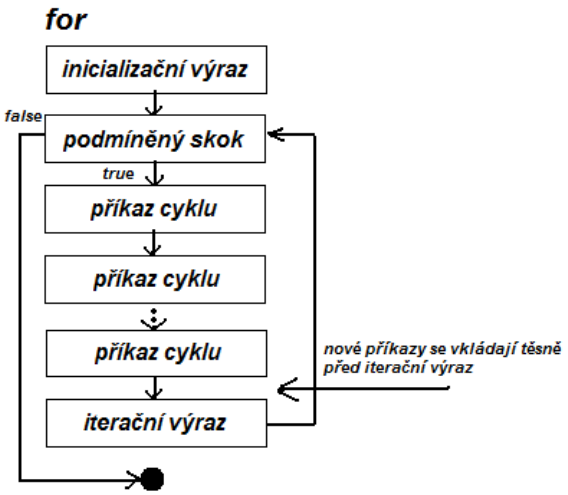
Obrázek 9: Struktura kódu if-else



Obrázek 10: Struktura kódu while



Obrázek 11: Struktura kódu do-while



Obrázek 12: Struktura kódu for

KodFor, obrázek 12 Cyklu **for**, tedy cyklus se známým počtem opakování je nejsložitější. Obsahuje dva výrazy inicializační a iterační a jednu terminální podmínku. Inicializační výraz se vyhodnocuje jako první, následuje terminální podmínka (příkaz podmíněný skok), který ukazuje na konec celého cyklu. Poté následují příkazy uvnitř cyklu a na konci cyklu je iterační výraz. Po vykonání iteračního výrazu je řízení přesměrováno zpět na terminální podmínku. Příkazy se vkládají vždy těsně před iterační výraz.

KodFunkce Kód funkce je reprezentován třídou `KodFunkce`, jedná se o obyčejný lineární kód doplněný o rozhraní pro tvorbu algoritmů. Toto rozhraní je popsáno v příloze A.

6 Návrh a implementace jazykových elementů

Jazykovými elementy jsou myšleny např. datové typy, operace, triviální funkce, různé proměnné odkazy apod.. Všechny jsou implementovány v balíku jazyky.

6.1 Třída Jazyk

Aplikace využívající aplikační rámec přistupuje k jazykovým elementům výhradně přes třídu Jazyk. V této třídě je definována sada statických operací a konstant pro definování všech použitých jazykových elementů. Třída Jazyk je vlastně aplikačním rozhraním pro celý balík jazyky. Jedná se o příklad využití návrhového vzoru Fasáda (Facade).

6.2 Řešení reprezentace datových typů

Každý algoritmus potřebuje během svého chodu uchovávat data (např. v proměnných). Aby bylo jasné co data obsahují je nutné, aby každý údaj měl přiřazený datový typ. Na datových typech údajů je závislé chování např. operací (není umožněno sčítaní znaku a čísla apod.). Datové typy jsou reprezentovány rozhraním IUdaj v balíku stroj. Implementace je pak realizována abstraktní třídou Udaj a jejími potomky. Třídní diagram je znázorněn na obrázku 13. Důležitá je metoda dejTypUdaje(), která vrací textový řetězec udávající název datového typu uloženého v údaji. Metodu si implementuje každý potomek sám podle toho, který datový typ zrovna obsahuje. Názvy datových typů nejsou nijak centrálně uchovávány ani kontrolovány. Je tedy na uživatelích aplikačního rámce, aby si hlídalo případné kolize názvu.

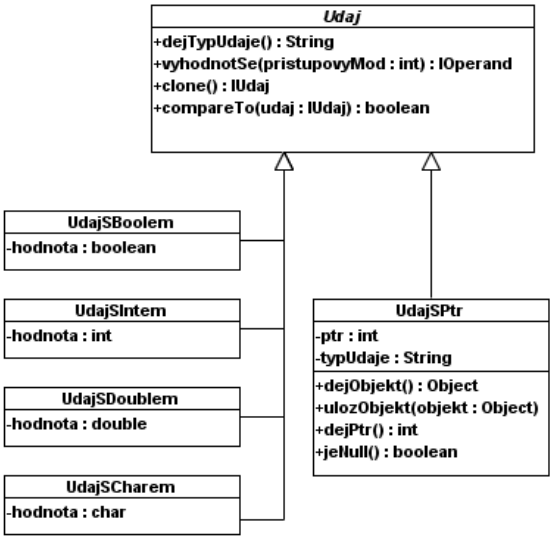
Vytváření nových datových typů je popsáno v příloze B.

6.2.1 Primitivní datové typy

Primitivních datových typů je v programovacím jazyce Java celá řádka. Jedná se o data přímo uložená v hašovací tabulce v paměti viz. 4.4. V aplikačním rámci jsou k dispozici pouze čtyři základní **double** na uchovávání čísel v plovoucí řadové čárce (třída UdajSDoublem, **int** na čísla s pevnou řadovou čárkou (třída UdajSIntem), **char** k uchovávání znaků (třída UdajSCharem) a **boolean** reprezentující logickou hodnotu (třída UdajsBoolem). Všechny uvedené třídy obalují primitivní datový typ a implementuji metody pro kopírování (clone()) a případně i pro porovnání (compareTo(), jeRovno()). Metodu compareTo() neimplementuje UdajSBoolem. Po kontrole pomocí metody dejTypUdaje(), můžeme údaj bezpečně přetypovat a získat tak z něj uložená data.

6.2.2 Složené datové typy

Složené datové typy jsou realizovány třídou UdajSPtr, která drží odkaz v podobě integeru na umístění v haldě v paměti viz.4.4. Třída umožňuje skrz tento odkaz pracovat s uloženým objektem (dejObjekt(), ulozObjekt(), který je typu Object). Do složeného datového typu se dá tedy uložit v podstatě jakákoli třída. V případě, že odkaz neukazuje nikam



Obrázek 13: Třídní diagram údajů

(má hodnotu -1), je v údaji uloženo **null**, tento stav lze otestovat metodou jeNull(). Mezi složené datové typy patří i pole a String.

6.2.3 Pole

Pole je řešeno jako složený datový typ. Implementace je realizována ve třídě Pole. Jedná se o obalení jednorozměrného pole údajů, které si navíc udržuje strukturu, je tedy možno pomocí něj snadno vytvářet i vícerozměrná pole. Konstruktor má dva parametry, název ukládaných datových typů (do pole lze uložit pouze jeden typ údajů) a počet rozměrů. Zpětně lze typ údaje získat metodou dejTypUdaje() a počet rozměru dejRozmery(). Inicializace se provádí pomocí metody inicializujPole(), která jako parametry očekává sadu čísel označující velikost jednotlivých rozměrů. ulozUdaj() a nactiUdaj() jsou poté metody sloužící k ukládání respektive čtení údajů z pole.

Pří získávání a ukládání údajů se adresa vypočte na základě indexů předaných v parametru přístupové metody a vnitřní struktury pole. Název datového typu pole je roven názvu ukládaného datového typu doplněného o příslušný počet hranatých závorek (např. dvourozměrné pole intů má název int[][]).

6.3 Řešení operací

Operace jsou součástí výrazů a určeným způsobem transformují jeden či dva vstupní operandy v jeden výstupní. Příkladem operace je například sčítání (5+6), kdy vstupními operandy jsou sčítance (5 a 6) a výstupním operandem pak součet (11). Operace může mít i postranní efekt například operace přiřazení (a=5) jako taková vrací výstupní operand číslo 5, během provádění operace však provádí zápis do paměti, kdy se do proměnné ''a''

název třídy	poskytované operace
OperaceIncDec	inkrementace,dekrementace
BitoveOperace	posun vlevo,posun vpravo, posun vpravo s rozšířením
LogickeOperace	logický součin, logický součet, exkluzivní součet
UnarniOperace	unární plus/minus, negace, bitový doplněk
AritmetickeOperace	sčítání,odčítání,dělení,násobení,modulo
PorovnavaciOperace	menší, menší rovno, rovno, nerovno, větší, větší rovno
PodminkoveOperace	podmínkový součin, podmínkový součet
PřiřazovacíOperace	přiřazení

Tabulka 3: Rozvržení operací do tříd

uloží číslo 5. Tento efekt nemusí být vždy žádoucí, například při prohlédnutí výsledku nějakého výrazu, bez toho aby bylo zapsáno do paměti.

6.3.1 Implementace operací

Základní implementace operací je realizována třídou Operace, která implementuje rozhraní IOperace. Třída poskytuje základní služby jako uchovávání vlastností operace (priorita, asociativita, binární/unární). Metoda proved() však zůstává abstraktní, a každý potomek si ji musí sám do implementovat.

Při provádění operací se nejprve zkontrolují datové typy operandů, vyhovují-li předpisu, operace se provede. Operace samozřejmě může mít těchto předpisu několik (např. sčítání). Nenajde-li se žádný vyhovující předpis provádění skončí vyjímkou.

Protože je v jazyce Java operací relativně hodně (viz. tabulka C, jsou shlukovány do tříd podle smyslu. Rozložení je uvedeno v tabulce 3).

6.4 Odkaz

Odkazy jsou chápány jako odkazy na určité údaje. Odkazem může být proměnná, proměnná v poli tedy určená indexy v hranatých závorkách [], nebo nějaký atribut složitějšího objektu (např. velikost pole). Odkazy se používají ve výrazech.

Odkaz je implementován ve třídě Odkaz. Nejdůležitějšími metodami jsou metody přístupové. Číst údaj z odkazu se dá pomocí metody vyhodnotSe(), ukládat poté metodami ulozUdaj() pro příme uložení údaje nebo ulozUdajZOdkazu() pro uložení údaje získaného z jiného odkazu. Díky tomu, že potomci této třídy mohou udržovat jakékoli informace navíc a mohou přetížit přístupové metody, stává se z odkazu velice flexibilní nástroj.

V aplikačním rámci jsou implementovány tři druhy odkazu jednoduchá proměnná (třída Promenna), proměnná získávána z pole (třída PromennaVPoli) a velikost pole (třída VelikostPole).

Jak používat odkazy popřípadě jak rozšířit aplikační rámec o své vlastní odkazy je popsáno B.

název třídy	popis	příklad zápisu v Javě
NovePole	inicializace pole	**new int**[4][4]
NovyString	konstruktor textového řetězce	‘Ahoj světe‘
FunkceSetridenePole	vytvoření setříděného pole intů	setridenePole(10)
FunkceNahodnePole	vytvoření nesetříděného pole intů	nahodnePole(10)
FunkceRandom	generování náhodného intu	random()

Tabulka 4: Seznam triviálních funkcí v aplikačním rámci

6.4.1 Promenna

Třída reprezentuje obyčejnou proměnnou. V konstruktoru se předá název proměnné a tím je vše hotovo. Přístupové metody poté spolupracují s pamětí a umožňují tak ukládat či načítat uložené údaje. Během ukládání údajů jsou vyvolány události doPromenneUlozeno respektive doPromenneUlozenoZOdkazu viz tabulka 1.

6.4.2 PromennaVPoli

Tato třída reprezentuje buňku v poli (v Javě se s nimi pracuje pomocí hranatých závorek a[0]=5 apod.). Na rozdíl od jednoduché proměnné se této třídě v předává v konstruktoru název pole a seznam výrazů reprezentující indexy. Během ukládání či čtení údajů se nejprve získá z paměti příslušné pole, proběhne vyhodnocení indexů a následně se metodami definovanými ve třídě Pole získá nebo uloží příslušná hodnota na příslušné místo. Během ukládání údajů jsou vyvolány události doPromenneVPoliUlozeno respektive doPromenneVPoliUlozenoZOdkazu viz tabulka 1

6.4.3 velikost pole

Velikost pole je poslední odkazem v aplikačním rámci. Reprezentuje zápis pole.length; nebo pole []. length, kde pole je nějaké proměnná typu pole. Na rozdíl od obou předchozích je tento odkaz určen jen pro čtení, pokus o zápis skončí vždy vyjímkou. Odkaz je definován názvem pole a rozměrem jehož velikost chceme získat.

6.5 Triviální funkce

Volání triviálních tedy nerozepsaných funkcí jsou reprezentovány potomky třídy Funkce. V konstruktoru se předává její název, příznak zda-li je konstrukční (konstrukční se chápe tak, že se při prezentaci automaticky před ní vloží klíčové slovo **new**), a seznam výrazů jako parametry funkcí. Vlastní provádění je pak realizováno v potomcích. Provedení každé funkce vyvolá událost funkceZavolana viz tabulka 1

Seznam dostupných funkcí je uveden v tabulce 4. Jak vytvářet své vlastní funkce je pak popsáno v příloze B.

6.6 Volání netriviálních funkcí

Volání netriviálních tedy rozepsaných funkcí je reprezentováno třídou volaniFunkce(). Parametry jsou odkaz na KodFunkce a seznam Vyraz, který určuje parametry předané funkci. Volán netriviálních funkcí je vždy součástí výrazu.

7 Návrh a implementace vizualizační části

Vizualizace zapsaných algoritmů je oddělená od ostatních části aplikačního rámce, informace získává odposloucháváním příchozích událostí ze třídy stroj.Stroj viz. tabulka 1. Vizualizace se dělí na dvě hlavní části, na část zobrazení zdrojového kódu a část vizualizace spouštěných algoritmů. Implementace je obsažena v balíku prezentace a je z velké části založená na komponentách javax.Swing.

7.1 Vizualizace zdrojového kódu

Tato část má na starosti zobrazení zdrojového kódu, jeho správné odsazení, zvýraznění klíčových slov, a označení právě vykonávaných příkazů. Mimo to obsahuje rychlé zobrazení hodnot proměnných a výsledků výrazů, a to ihned po najetí kurzoru nad příslušnou část.

7.1.1 Vizualizace jednoho členu

Třída ProhlizecClenaVyrazu je základem vizualizace jednoho členu (název proměnné, výraz apod.). Třída rozšiřuje JComponent o metodu aktualizuj(), která vyhodnotí vizualizovaný výraz nebo operand a výsledek zapíše do bublinkové nápovědy, tak aby po najetí kurzoru byl výsledek k dispozici. Důležité je, že vyhodnocení se provádí v módu pouze pro čtení, takže se neprovádí zápis do paměti, to umožňuje vyhodnotit např. a++ aniž by proměnná a byla změněna.

Konstrukce členů výrazu se provádí ve třídě TovarnaNaProhlizeceKodu, která obsahuje seznam statických metod určených k vytváření těchto prohlížečů. Tyto metody jsou volány z jednotlivých tříd určených k vizualizaci.

7.1.2 Vizualizace kódu

Vizualizace implementována v ProhlizecKodu na rozdíl od vizualizace jednoho člena provádí vizualizaci celých příkazů a programových bloků. Třída ProhlizecKodu rozšiřuje JPanel o metody pro vkládání zobrazovaných příkazu a metodu pro porovnání příkazů uvnitř vizualizovaného kódu. Metoda pro porovnání je důležitá v tom, že při procházení kódu se aktuální příkaz označuje.

Jednotlivé prohlížeče kódu se stejně jako prohlížeče členu výrazu realizují ve třídě TovarnaNaProhlizeceKodu.

7.1.3 Třída ProhlizecHlavnihoKodu

Tato třída rozšiřuje JComponent a zastřešuje celou vizualizaci zdrojového kódu. Vizualizovaný kód se načítá metodou nactiKod, dále je komponentě možno nastavit základní vlastnosti jako font, barvu fontu, barvu pozadí, podbarvení prováděného příkazu. Komponenta je v konstruktoru automaticky připojena jako posluchač ke stroji a odchytává události *programNacten*, *programUkoncen* a *pripravPrikaz*, poslední jmenovaná událost říká,

který příkaz bude v dalším kroku vykonán a díky tomu jej prohlížeč může podbarvit a zvýraznit.

7.2 Vizualizace algoritmu

Tato část má na starosti vlastní zobrazení chování algoritmu, jako zobrazení a přesouvání proměnných zobrazení polí apod.

7.2.1 Třída ProhlizeciVrstva

Prohlížecí vrstva tvoří základní vrstvu celé vizualizace. Třída rozšiřuje JLayeredPane což umožňuje vkládat jiné komponenty na základě absolutních souřadnic a do jednotlivých vrstev. Třídu jsem navrhl tak, aby bylo přesouvání jednotlivých komponent co nejjednodušší na použití. Ve třídě je definován seznam úloh. Úlohou se rozumí přesun komponenty z místa na místo, popř. její vyjmutí po ukončení přesunu. Po nadefinování úloh lze animaci spustit, což spočívá ve spuštění vlákna jež definovaným postupem posouvá všechny určené komponenty, vykonává tedy úlohy. Komponent lze posouvat více najednou, každé posunutí se rovna jedné úloze. Po přesunutí všech komponent je vyvolána událost krok ukončen, kterou informuje své posluchače o ukončení animace.

Přesouvat komponenty lze po přímce, lomené čáře nebo po Beziérově křivce. Důležité vlastnosti prohlížecí vrstvy jsou potom mód přesunutí, který může být buď s konstantním počtem kroků nebo s konstantní délkou kroku, je-li konstantní počet kroku komponenty se budou přesouvat různou rychlostí, ale přesun skončí v jeden okamžik. Při přesunu s konstantní délkou kroku se komponenty pohybují stejnou rychlostí ale dorazí do cíle v různou dobu, na základě uražené vzdálenosti. Poslední důležitou vlastností je čas na jeden krok.

7.2.2 Třída ProhlizecAlgoritmu

Prohlížeč algoritmu obsahuje prohlížecí vrstvu a poskytuje rozhraní ven aplikacím používajícím tento aplikační rámec. Také nastaví prohlížecí vrstvu do módu s konstantním velikost kroků, která bude činit 1 pixel a časem určeným na krok 5 ms.

Dále obsahuje seznam animátoru což jsou entity zobrazované uvnitř vrstvy, jedná se o vizualizované komponenty, které musí o sobě navzájem vědět. Např. jednoduchý popisek umístěny na prohlížecí vrstvu nemusí být animátor, protože nějak nespolupracuje s ostatními, animátor proměnné však již animátorem být musí, protože je pravděpodobné, že se budou mezi proměnnými provádět přesuny. Animátory rozšiřují rozhraní IAnimator.

7.2.3 Animátor proměnné

Prvním použitým animátorem je animátor proměnné, sloužící k vizualizaci jedné jednoduché proměnné. Implementován je ve třídě AnimatorPromenne. Animátor je definován názvem proměnné které vizualizuje, jeho vzhled pak třídou SchemaAnimatoruPromenne.

název události	**chování**
deklaracePromenneProvedena	zobrazení proměnné
bunkyVPametiZrusena	zrušení proměnné
doPromenneUlozeno	přepsání hodnoty
doPromenneUlozenoZOdkazu	přesunutí hodnoty

Tabulka 5: Události na něž reaguje animátor proměnné

název události	**chování**
incializacePole	konstrukce a zobrazení
deklaracePromenneProvedena	zobrazení ukazatele (šipky)
bunkyVPametiZrusena	skrytí ukazatele (šipky)
doPromenneUlozeno	přesunutí ukazatele (šipky)
doPromenneUlozenoZOdkazu	přesunutí ukazatele (šipky)
doPromenneVPoliUlozeno	přepsání příslušné hodnoty
doPromenneVPoliUlozenoZOdkazu	přesunutí příslušné hodnoty

Tabulka 6: Události na něž reaguje animátor pole

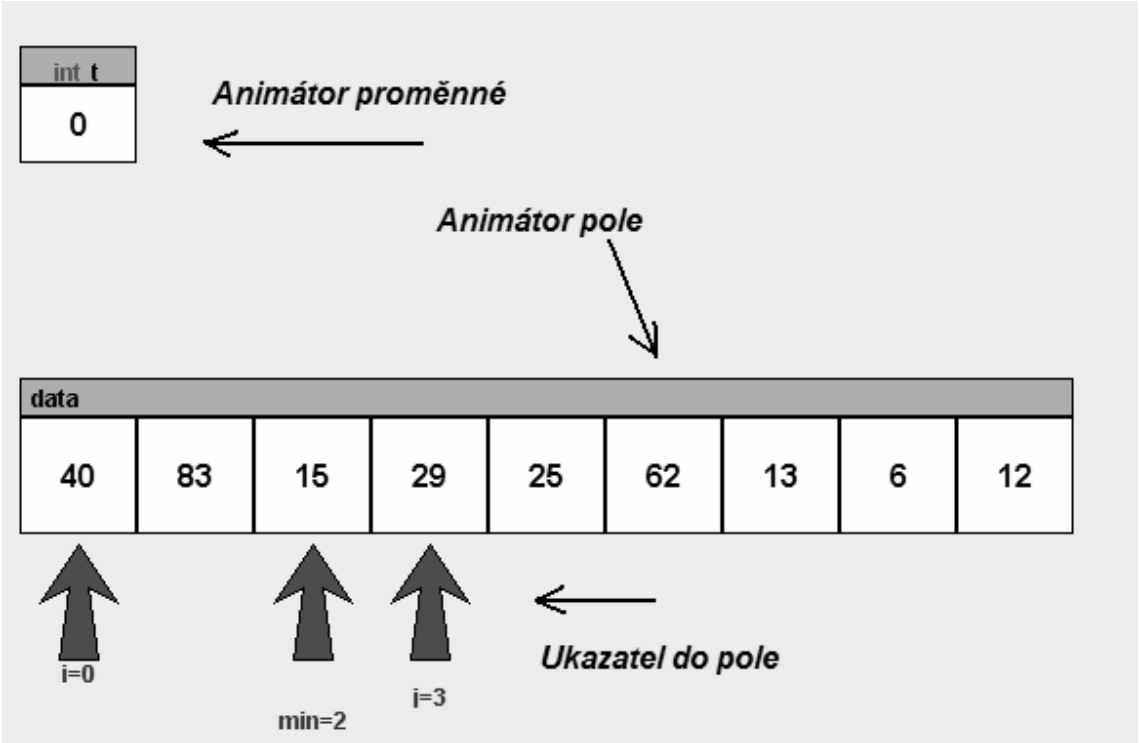
Skládá se ze dvou panelu JPanel a třech popisků JLabel. Panely reprezentují hlavičku a tělo. V hlavičce jsou potom údaje o typu a názvu proměnné v samotném těle potom hodnota proměnné.

Události na nějž animátor proměnné reaguje jsou popsány v tabulce 5.

7.2.4 Animátor pole

Animátor pole slouží k vizualizaci jednoduchého pole. Implementace je realizována ve třídě AnimatorPole. Stejně jako předchozí animátor je i tento definován názvem pole, které obsahuje. Navíc lze přidat ukazatele, tedy šipky ukazující na jednotlivé buňky pole, šipka je definována názvy proměnných na které reaguje a výrazem jež vyhodnocuje. Vzhled je určen třídou SchemaAnimatoruPole.

Reakce na události jsou shrnuty v tabulce 6



Obrázek 14: Ukázka vzhledu animátorů

8 Rozšíření funkčnosti rámce

Rozšířením aplikačního rámce je chápáno hlavně možnost doplňování jazykových elementů. Velice snadno lze doplnit jednoduché triviální funkce a to rozšířením třídy jazyk.Funkce. Operace lze definovat rozšířením třídy jazyk.Operace, operace lze i přetěžovat. Při řešení výrazu volá systém operace definované ve třídě jazyk.Jazyk. Tato třída obsahuje statické metody, které umožňují obsažené funkce přepsat a tak změnit chování operátorů. Jednoduchým způsobem lze definovat různé podpůrné třídy s veřejnými atributy a to pomocí třídy jazyk.Odkaz. Možnosti i s ukázkou jsou popsány v příloze B.

Dalším možnostmi rozšiřování je možnost měnit pohled na provádění kódu, tedy vizualizaci. Vzhledem k tomu, že vizualizační vrstva přijímá komponenty javax.Swing jsou v tomto směru možnosti velmi bohaté s jistými úpravami by bylo možno vytvářet i interaktivní prezentace umožňující uživatelům aktivní zásahy do prováděného algoritmu.

Základním nedostatkem je nemožnost sestavit kód bez rekompilace zdrojových souborů. Tento nedostatek by mohl být vyřešen doplněním systému o objekty starajíce se o načítání kódu z externích zdrojů např. textového souboru.

Jistě by bylo vhodné při sestavování kódu kontrolovat syntaxi i vzájemné vztahy objektu. Například zda-li se nepoužívá ještě nedeklarovaná proměnná apod.

Další věcí, která by zvýšila kvalitu aplikačního rámce je sjednotit přístup a chování vizualizačních prvků a případná implementace designera umožňujícího snadnou tvorbu prezentace algoritmu.

9 Závěr

V této bakalářské práci jsem navrhnul a na implementoval aplikační rámec umožňující automatickou animaci průběhu algoritmů. Aplikační rámec jsem navrhnul s ohledem na jednoduchou použitelnost a rozšiřitelnost. Během celé doby jsem zhodnocoval jak teoretické znalosti tak praktické dovednosti nabyté během mého studia.

Podařilo se mi splnit základní požadavky na snadnou rozšiřitelnost. Pomocí tříd aplikačního rámce lze snadno nadefinovat zdrojový kód animovaného algoritmu a následně jej rozanimovat, přičemž vše se děje automaticky bez zásahu vnější aplikace. Součástí práce je i implementace aplikace demonstrující použíti rámce a to jak definice zdrojového kódu, definice prezentace tak i spouštění a řízení probíhajícího kódu. Mimo drobných nedostatků se hlavně nepodařilo vyvinout způsob načítání zdrojového kódu z externího souboru.

S nasazením se počítá hlavně při výuce základu algoritmizace. Do budoucna bych rád na tomto projektu pracoval a rozvíjel jej tak, aby mnohem lépe splňoval praktické potřeby svých uživatelů, at' studentů či vyučujících.

Během mé práce jsem získal neocenitelné zkušenosti s vývojem rozsáhlejšího a komplexnějšího projektu.

10 Literatura

[1] SEMECKÝ, Jiří. *Naučte se Javu : operátory a řídící příkazy.* Interval [online]. 2002. Dostupný z WWW: <http://interval.cz/clanky/naucte-se-javu-operatory-a-ridici-prikazy/>.

[2] *JDK 6 Documentation [online]*. Sun Microsystems, c2006. Dostupný z WWW: <http://java.sun.com/javase/6/docs/>.

[3] prof. PELIKÁN, Jan. *Stromové struktury.* Matematické základy informatiky [online]. 2006. Dostupný z WWW: <http://hroch.sk/skola/mzi/6_stromy.doc>.

[4] PLACKOVÁ, Eva. *Framework pro animaci algoritmů.* [s.l.], 2008. 30 s. Vysoká škola Báňská - Technická univerzita Ostrava. Vedoucí bakalářské práce Ing. David Ježek.

A Použití aplikačního rámce

Programátorské použití spočívá ve třech krocích, definice zdrojového kódu, definování vizualizace a následné provázání a spouštění. Všechny vyjmenované kroky v této příloze důkladně popíšu.

A.1 Sestavení zdrojového kódu

Postup sestavení zdrojového kódu se skládá ze dvou kroků. Sestavení jednotlivých funkcí a sestavení programu. Upozorňuji, že při sestavování kódu nedochází téměř k žádným kontrolám (až na kontrolu správné syntaxe výrazu), použije-li se např. proměnná, která nebyla před tím deklarována nebo se zavolá neexistující funkce, aplikační rámec to nepozná a k chybě dojde až při spuštění špatně napsaného algoritmu.

A.1.1 Výrazy

Výraz patří k základním konstrukcím. Protože se jedná o základní věc, která je použitá ve všech příkazech vysvětlím pravidla sestavování výrazu hned na začátek. Řeknu-li výraz mám na mysli něco jako `a = a + 2;`, `pole[x] = 4;` nebo také složitější `pole = new int [a+1][random(6++)]`. Výraz je reprezentován třídou `Vyraz`. Konstrukce se provádí pomocí statické metody `vyraz` ve třídě `jazyk.Jazyk`. Metoda přijímá buď jeden nebo více členů výrazu. Přičemž členy výrazu mohou být:

- Konstanty
- Operace
- Proměnné
- Volání triviálních funkcí
- Volání uživatelem definovaných funkcí
- Jiné výrazy

Konstanty
Konstanty jsou jednoduchá čísla, popř. znaky nebo i řetězec. Konstantu lze vytvořit statickou metodou `udaj`, hodnotu konstanty pak určuje parametr.

```
udaj(0); // konstanta typu int, hodnota 0
udaj(true); // konstanta typu boolean, hodnota true
udaj(−4.4); // konstanta typu double, hodnota −4.4
udaj('c'); // konstanta typu char, hodnota c
string("ahoj"); // konstanta typu string, hodnota "ahoj"
udajNull("String"); // konstanta typu string, hodnota null
```

Je-li potřeba mít konstantu ve formě výrazu lze použít metody stejného názvu rozšířené o písmeno v. Konstanta pak bude uzávorkovaná.

```
udajv(0); // konstanta typu int, hodnota 0
udajv(true); // konstanta typu boolean, hodnota true
udajv(−4.4); // konstanta typu double, hodnota −4.4
udajv('c'); // konstanta typu char, hodnota c
stringv("ahoj"); // konstanta typu string, hodnota "ahoj"
```

V případě konstanty **null** je nutné udávat název datového typu do kterého bude vkládána!

Operace
Pro navrácení operace je potřeba zavolat příslušnou statickou metodu opět z třídy jazyk.Jazyk. Použitelné operace jsou zobrazeny v tabulce 7. Vedle názvu operace je uveden operátor, který ji představuje.

Proměnné
Proměnných existují dva typy, jednoduché proměnné a proměnné v poli. Pro vytváření proměnných existují dvě metody

```
promenna("a"); // vrátí proměnnou "a" jako odkaz
promennav("a"); // vrátí proměnnou "a" jako výraz
```

Je-li třeba odkázat na proměnnou v poli můžeme ji indexovat obyčejnými čísly

```
promenna("pole", 0, 0); // vrátí proměnnou "pole[0][0]" jako odkaz
promennav("pole", 0, 0); // vrátí proměnnou "pole[0][0]" jako výraz
```

Nebo můžeme indexovat pomocí jiných výrazů.

```
promenna("pole", promennav(a), udajv(0))); // vrátí proměnnou "pole[a][0]" jako odkaz
promennav("pole", udajv(0), promennav(b)); // vrátí proměnnou "pole[0][b]" jako výraz
```

Počet indexů samozřejmě závisí na počtu rozměrů daného pole.

Volání triviálních funkcí
V tomto případě se jedná o volání vestavěných funkcí vracející hodnotu přímo. Je-li ve výrazu takové volání (bez toho aby v něm současně bylo volání netriviální funkce) výraz se vyhodnotí přímo. Použitelné funkce a jejich metody jsou zapsány v tabulce 8

Volání složených funkcí
V tomto případě se jedná o volání složených uživatelem definovaných funkcí, tedy těch jež jsou sestaveny pomocí třídy kody.KodFunkce. Po zavolání takové to funkce je nutné

scitani(+)	dekrementace(−−)	podminkovySoucin(&&)
odcitani(−)	jeRovno(==)	podminkovySoucet(||)
nasobeni(∗)	jeMensi(<)	logickySoucit(&)
deleni(−)	jeVetsi(>)	logickySoucet(|)
modus(%)	jeMensiNeboRovno(<=)	exkluzivniSoucet(^)
prirazeni(=)	jeVetsiNeboRovno(>=)	posunVlevo(<<)
inkrementace(++)	neniRovno(! =)	posunVpravo(>>)

Tabulka 7: Metody na dosazení operací jazyk.Jazyk

název metody	**popis**	**parametry**	**popis parametrů**
novePole	inic. pole	string, int...	typ, vel. rozměrů
novePole	inci. pole	string, Vyraz...	typ, vel. rozměrů
random	generátor náhodného čísla	int	max. vel. náhodného čísla
setridenePole	vrací setřítědé pole intů	int	vel. pole
nahodnePole	vrací náhodné pole intů	int	vel. pole

Tabulka 8: Seznam metod pro dosazení triviálních funkcí Jazyk.jazyk

předat případné parametry. Metody kterými jde do volání dosadit výrazy jsou funkce (KodFunkce, Vyraz... a funkcev(KodFunkce, Vyraz..., prvním parametrem je odkaz na příslušný kód funkce a druhým potom proměnný seznam výrazu předaných do funkce jako parametry. Počet výrazu musí být roven počtu parametru volané funkce. Ukázka jsou potom zobrazeny níže při vysvětlování sestavení funkcí a programu.

```
 // 1) Výraz a++ by se zapsal takto:
Vyraz v1 = vyraz(promenna("a"), inkrementace());

 // 2) Výraz hlava = hlava % data.length by se zapsal takto (proměnná data je typu pole):
Vyraz v2= vyraz(promenna("hlava"), prirazeni(), promenna("hlava"), modus(), velikostPole("
    data"));

// 3) Výraz data = new String[random(10)] by se zapsal takto:
Vyraz v3 = vyraz(promenna("data"), prirazeni(), novePole("String", random(10)));

// 4) Výraz a + (5−c) ∗ ( a + (b / 2)) by se zapsal takto:
Vyraz v4 = vyraz(promenna("a"), scitani(),
              vyraz(udaj(5), odcitani(), promenna("c")),
              nasobeni(),
              vyraz(promenna("a"), scitani(),
                  vyraz(promenna("b"), deleni(), udaj(2))));

// 5) Výraz c == (a < b) by zapsal takto:
Vyraz v5 = vyraz(promenna("c"), jeRovno(), vyraz(promenna("a"), jeMensi(), promenna("b")));
```

Výpis 2: Ukázky sestavení výrazů

A.1.2 Definice funkce

Jednotlivé funkce jsou prezentovány třídou KodFunkce v balíku kody. Při sestavování se musí nejprve definovat hlavička funkce, tedy to jak se funkce bude jmenovat, jaké bude přijímat parametry a případně jakého typu bude návratová hodnota. Definice hlavičky se provádí přímo v konstruktoru třídy a má několik přetížení. Jednoduchá funkce s názvem *main*, která je bez parametru a nevrací žádnou návratovou hodnotu by se vytvořila takto:

```
KodFunkce fMain = new KodFunkce("main")
```

V případě bezparametrické funkce, která ale již vrací nějakou hodnotu použijeme přetížený konstruktor s dvěma parametry. Prvním parametrem je název funkce, druhým název datového typu návratové hodnoty.

```
KodFunkce fPi = new KodFunkce("pi", "double");
```

Tato funkce se tedy nazývá "pi" a vrací hodnotu double. Aby funkce přijímala jeden parametr a zároveň vracela návratovou hodnotu je nutné použít tento konstruktor:

```
KodFunkce fFaktorial = new KodFunkce("faktorial", "int", "int", "n");
```

Konstruktor má již čtyři parametry, název funkce, název datového typu návratové hodnoty, název datového typu parametru a samotný název parametru. Název parametru je v podstatě název proměnné, pod kterým bude předaná hodnota ve funkci vystupovat. Předchozí funkce vrací tedy hodnotu typu int a potřebuje jeden parametr typu int, který se bude jmenovat "n". Má-li mít funkce více parametru použijeme místo dvou posledních parametrů seznamy.

```
KodFunkce fMocneni = new KodFunkce("umocni", "double", {"double", "int"}, {"z", "exp"});
```

Jeden seznam představuje názvy datových typů, druhý seznam potom názvy parametrů. Funkce bude tedy přijímat dva parametry, parametr "z" bude typu double, zatímco parametr "exp" typu int. Je nutné aby oba seznamy byly stejně dlouhé. Při definování funkce bez návratové hodnoty se vypustí druhý parametr, návratový typ bude automaticky doplněn na void. Dvě poslední ukázky nyní bez návratových hodnot.

```
KodFunkce fFaktorial = new KodFunkce("faktorial", "int", "n");
KodFunkce fMocneni = new KodFunkce("umocni", {"double", "int"}, {"z", "exp"});
```

Teď, když je hlavička funkce hotova lze do jejího těla přidávat jednotlivé příkazy(výraz, deklaraci proměnné a příkaz **return**).

A.1.3 Vložení výrazu

Vložení výrazu do těla funkce se provádí jednoduchým voláním.

```
funkce.vlozVyraz(vyraz(promenna("a"), prirazeni(), udaj(0)));
```

Jak sestavit výraz je popsáno v kapitole A.1.1

A.1.4 Deklarace proměnné

Deklarace proměnné je nutno vložit ještě před samotným používáním proměnné. Na rozdíl od pravidel Javy, je zde nutno každou nově deklarovanou proměnnou ihned inicializovat. Deklarace proměnné se tedy skládá z názvu datového typu proměnné s názvu samotné proměnné a inicializační hodnoty. Ve třídě jsou k dispozici dvě metody.

```
funkce.vlozDeklaraciPromenne("int", "a", udaj(0));
funkce.vlozDeklaraciPromenne("double", "b", vyraz(promenna("a"));
```

První metoda přiřazuje proměnné "a", která je typu int údaj s hodnotou 0. Druhá potom deklaruje proměnnou "b" jako double a přiřazuje ji výraz. Jak psát údaje a výrazy je vysvětleno v kapitole A.1.1. Při deklaraci pole stačí rozšířit název datového typu o příslušný počet závorek.

```
funkce.vlozDeklaraciPromenne("int[][]", "matice", udajNull("int [][] "));
funkce.vlozDeklaraciPromenne("String[]", "pole", novePole("String", 5);
```

První ukázka deklaruje dvourozměrné pole "matice" a přiřadí ji hodnotu null. Druhá ukázka potom deklaruje pole stringů "pole" a inicializuje jej na 5 prvků.

A.1.5 Příkaz return

Příkaz return slouží k ukončení funkce. Může obsahovat návratovou hodnotu. Implicitně je na konec každé funkce vložen příkaz return bez návratové hodnoty. Vložit do těla funkce příkaz return je nutné tehdy, když je potřeba vracet nějakou návratovou hodnotu nebo když je potřeba funkci předčasně ukončit. Vložení příkazu return s návratovou hodnotou a+5.

```
funkce.vlozReturn(vyraz(promenna("a"), scitani(), udaj(5)));
```

Vložení příkazu return bez návratové hodnoty se pak provede takto.

```
funkce.vlozReturn();
```

Upozornění: Funkce nějak neporovnávají návratové hodnoty s použitými příkazy return, v případě špatně definovaného kódu není zaručeno správné chování ani vyhození vyjímky.

A.1.6 Větvení a cykly

Větvení a cykly se vkládají stejným způsobem jako předchozí příklady. Bude-li třeba blok kódu (větev nebo cyklus) ukončit provede se vložení konce bloku kódu a to metodou

```
vlozKonecBloku()
```

Vložení větvení
Větvení je reprezentováno bloky **if** a **if**−**else**. Větvení je definovanou pouze rozhodující podmínkou, tato podmínka se vkládá ve formě výrazu vracející hodnotu boolean. Příklad neúplného větvení.

```
funkce.vlozIf(vyraz(promenna("a"), jeRovno(), udaj(0)));
funkce.vlozVyraz(promenna("a"), prirazeni(), udaj(1));
funkce.vlozKonecBloku();
```

Příklad sestavení úplného rozhodovacího bloku, včetně větve else.

```
funkce.vlozIf(vyraz(promenna("a"), jeRovno(), udaj(0)));
funkce.vlozVyraz(promenna("a"), prirazeni(), udaj(1));
funkce.vlozElse()
funkce.vlozVyraz(promenna("a"), prirazeni(), udaj(2));
funkce.vlozKonecBloku();
```

Vložení cyklu while
Vložení cyklu **while** je obdobné jako vkládání větvení. Je definováno jen ukončující podmínkou. Ukázka vložení jednoduchého cyklu while.

```
funkce.vlozWhile(vyraz(promenna("a"), jeVetsi(), udaj(5)));
funkce.vlozVyraz(promenna("a"), inkrementace());
funkce.vlozKonecBloku();
```

Vložení cyklu do-while
Vložení cyklu **do**−**while** je obdobné jako vkládání předchozích bloků. Opět je definováno pouze ukončující podmínkou. Rozdílem oproti Javě je, že se podmínka zadává už u klíčového slova **do**. Ukázka.

```
funkce.vlozDoWhile(vyraz(promenna("a"), jeVetsi(), udaj(5)));
funkce.vlozVyraz(promenna("a"), inkrementace());
```

```
funkce.vlozKonecBloku();
```

Vložení cyklu for
Vkládání cyklu **for** je mírně komplikovanější. Hlavičku cyklu lze vložit metodou

```
funkce.vlozFor(vyraz(promenna("a"), prirazeni(), udaj(10)), vyraz(promenna("a"), jeVetsi(),
udaj(0)), vyraz(promenna("a"), dekrementace()));
```

Kde parametry jsou postupně incializační výraz, terminální podmínka a iterační výraz. V případě potřeby deklarovat proměnnou v inicializační části deklarovat je nutné použít přetížení metody.

```
funkce.vlozFor("int", "i", udajv(0), vyraz(promenna("i"), jeMensi(), udaj(5)),
vyraz(promenna("i"), inkrementace()));
```

první řetězec je název datového typu nově deklarované proměnné, druhý řetězec je její jméno, třetí je inicializační výraz, čtvrtý a pátý parametr je stejný jako v předchozím případě.

A.1.7 Kompletace programu

Teď, když jsou vytvořeny všechny funkce (jedna funkce = jeden objekt KodFunkce), je možno je dát dohromady. Celý program je reprezentován třídou kody.KodPrograma(). Vložení funkce do programu se provádí metodou vlozFunkci(KodFunkce), kde parametr je konkrétní již sestavená funkce. Důležité je, že v programu se smí nacházet právě jedna funkce s názvem *main*. Tato funkce vystupuje jako vstupní bod celého programu a je první volaná při spuštění programu.

Může se stát, že bude potřeba deklarovat globální proměnné. Taková deklarace se pak provádí metodou vlozGlobalniDeklaraciPromenne. Globální deklarace proměnných jsou spouštěny ještě před samotným voláním funkce *main*.

Ukázka kompletace programu je zobrazena ve výpise 3.

```
/*
 * Program Naplnění pole
 * globální proměnná data – pole řetězců
 * funkce: void naplnPole(String ret)
 *          naplní pole řetězců hodnotou obdrženou v parametru
 * funkce: void main()
 *           inicializuje pole, a nechá si jej naplnit hodnotou "ahoj"
 */

// Nejprve se vytvoří funkce naplnPole
KodFunkce fNaplnPole = new KodFunkce("naplnPole", "String", "ret");

// Kód se bude skládat s jednoho cyklu
fNaplnPole.vlozFor("int", "i", udajv(0),
                                vyraz(promenna("i"), jeMensi(), velikostPole("data")),
                                vyraz(promenna("i"), inkrementace();

  fNaplnPole.vlozVyraz(vyraz(promenna("data", promennav("i")),
                                prirazeni(),
                                promenna("ret");

fNaplnPole.vlozKonecBloku(); // ukončení cyklu
fNaplnPole.vlozKonecBloku(); // ukončení funkce

// Nyní se vytvoří funkce main
KodFunkce fMain = new KodFunkce("main");
// Inicializace pole o 10 řetězcích
fMain.vlozVyraz(vyraz(promenna("data"), prirazeni(), novePole("String", 10)));
// Zavolání funkce naplň pole s parametrem "ahoj";
fMain.vlozVyraz(volaniv(fNaplnPole, udajv("ahoj")));
fMain.vlozKonecBloku(); // ukončení funkce main

// Vytvoření programu NaplneniPole
KodProgramu program = new KodProgramu("NaplneniPole");

// Globální deklarace pole data
program.vlozGlobalniDeklaraci("String[]", "data", udajNull("String[]"));

// Vložíme obě funkce
program.vlozFunkci(fNaplnPole);
program.vlozFunkci(fMain);

// Program hotov
```

Výpis 3: Ukázka kompletace programu

A.2 Sestavení prezentace

Sestavení prezentace se skládá z definice umístění a chování jednotlivých vizualizačních prvků. Všechny prvky se vkládají do prohlížeče algoritmu (třída ProhlizecAlgoritmu). Zob-

razení proměnné se provede konstrukcí

```
new AnimatorPromenne("a", pA, new Point(20, 20));
```

První parametr je název zobrazené proměnné, druhým potom konkrétní prohlížeč, poslední potom místo vložení. Takže předchozí příkaz vytvoří prohlížeč proměnné "a" na prohlížeči "pA" na souřadnici 20,20. V případě, že vzhled animátoru proměnné nevyhovuje můžeme použít přetížení

```
new AnimatorPromenne("a", pA, new Point(20, 20), schema, true);
```

První tři parametry jsou shodné jako v předešlém případě, navíc je ale použito schéma. Posledním parametrem je příznak, zda-li má animátor reagovat na události. Schéma animátoru proměnné je realizováno ve třídě SchemaAnimatoruPromenne a jsou v něm definovány vlastnosti ovlivňující vzhled animátoru.

Zobrazení pole se provádí konstrukcí

```
new AnimatorPole("data", pA, new Point(20, 20));
```

Parametry mají stejný význam jako u animátoru proměnné, vytvoří se tedy animátor pole názvu "pole" v prohlížeči "pA" na pozici 20, 20. V případě změny vzhledu je nutné použít přetížení

```
new AnimatorPole("data", pA, new Point(20, 20), schema);
```

Stejně jako u animátoru proměnné se akorát přidá parametr definující schéma animátoru pole. Toto schéma je realizováno ve třídě SchemaAnimatoruPole. Mezi prvky vzhledu ve schématu se nachází i atribut smer, kterým lze ovlivnit orientaci pole. Možné hodnoty jsou SchemaAnimatoruPole.NA_STOJATO nebo SchemaAnimatoruPole.NA_LEZATO. Je-li potřeba k animátoru pole přiřadit i ukazatel, lze to provést voláním

```
aniPole.priradUkazatel("a", promennav("a"), true);
```

Příklad přiřadí animátoru pole šipku reagující na proměnnou "a", bude zobrazovat hodnotu proměnné "a" a smí při zrušení proměnné zmizet. První parametr tedy definuje název proměnné na jejíž změnu bude šipka reagovat, druhým potom výraz který se bude vyhodnocovat a poslední zda-li má šipka při zrušení proměnné zmizet. Šipka bude obarvena standardní barvou určenou schématem animátoru pole. Kdyby se přiřazení ukazatele provedlo takto

```
aniPole.priradUkazatel("b", vyraz(promennav("b"), scitani(), udaj(1)), false, GREEN);
```

Vytvoří se ukazatel reagující na změnu proměnné ”b”, nebude ale zobrazovat přímo její hodnotu nýbrž hodnotu o 1 větší tedy b+1, šipka nezmizí ani po vyjmutí proměnné ”b” z paměti a bude vyplněna zelenou barvou.

Kromě výše popsaných animátoru lze do prohlížeče algoritmu vložit jakoukoli proměnnou javax.Swing.

```
prohlizecAlgoritmu.vlozKomponentu(new JLabel(”Ahoj”), new Rectangle(0, 0, 50, 20));
```

Zobrazený příklad tedy vloží do levého horního rohu prohlížeče algoritmu nápis ”Ahoj”. Komponenta samozřejmě nemusí být JLabel, ale může být jakákoli, druhý parametr pak určuje přesné umístění včetně velikosti.

A.3 Spouštění kódu

Po sestavení kódu a sestavení prezentace jsou k dispozici tyto objekty

```
KodProgramu program; //třída se zdrojovým kódem algoritmu
ProhlizecAlgoritmu prohlizecAlg; // třída s prezentací
```

Dále si stačí vytvořit prohlížeč zdrojového kódu

```
ProhlizecHlavnihoKodu prohlizecKodu;
```

Kvůli tomu aby prohlížeč zdrojového kódu věděl kdy se ukončí animace je nutné provést provázání prohlížeče algoritmu s prohlížečem kódu.

```
prohlizecAlgoritmu.pridejPosluchace(prohlizecKodu);
```

Pomocí statické metody získáme instanci stroje

```
Stroj stroj = Stroj.dejStroj()
```

Vlastní načtení algoritmu se provede předáním kódu stroji

```
stroj.nactiProgram(program);
```

Je-li vše v pořádku, tak se po tomto zavolání zobrazí v prohlížeči kódu naformátovaný zdrojový kód algoritmu. Provedou se globální deklarace programu a v závislosti na nich i některé animátory proměnných. Program lze krokovat voláním

```
program.proved();
```

Vrátí-li metoda hodnota **true** program je u konce. V případě, že metoda *main* očekává

parametry lze je předat metodou

```
program.nastavParametry(Vyraz...);
```

Parametry jsou tedy reprezentovány polem výrazů. Ukončení algoritmu lze provést metodou

```
stroj.reset();
```

B Rozšíření aplikačního rámce

B.1 Rozšíření jazykových elementů

Jazykové elementy aplikačního rámce lze rozšířit o nové referenční typy, triviální funkce nebo odkazy. Přičemž odkazy mohou simulovat chování veřejných atributů tříd popřípadě jejich metod.

B.1.1 Vytvoření nového referenčního typu

Všechny objekty jsou v aplikačním rámci uloženy na haldě v paměti. V algoritmu se pak dá k nim dostat pomocí údaje UdajSPtr. Je-li potřeba vytvořit nový referenční typ stačí vytvořit libovolnou třídu, přičemž její atributy je nutné zaobalit do příslušného údaje (UdajSIntem,UdajSDoublem,UdajSBoolem,UdajSCharem popř. referenční typy UdajSPtr).

B.1.2 Vytvoření nové funkce

Je-li potřeba vytvořit novou funkci stačí vytvořit třídu dědící z abstraktní třídy jazyk.Funkce. Přičemž v konstruktoru se musí předat název funkce, informace zda-li se má před název vložit klíčové slovo **new** a také případné parametry funkce. Nutné je překrýt metodu vyhodnotSe(), kde je implementována celá logika funkce. Metoda vrací IUdaj, tedy nějaký abstraktní údaj. Zůstane-li metoda nepřekryta bude vždy vracet hodnotu null. Během provádění kódu funkce je nutno zavolat na předka vyhodnotSe() a to kvůli vyvolání události o zavolání funkce. Lze také překrýt standardní metodu toString(), jejíž hodnota je pak zobrazena v bublinkové nápovědě u funkce při prezentaci zdrojového kódu.

B.1.3 Vytvoření nového odkazu

Pro vytvoření nového odkazu je třeba dědit z abstraktní třídy jazyk.Odkaz. Zde je důležité překrytí metod ulozUdaj(), ulozUdajZOdkazu() a vyhodnotSe(). Zatímco první dvě metody slouží k uložení hodnoty do odkazu tak poslední metoda slouží k přečtení hodnoty z odkazu. Zde je důležité i překrytí standardní metody toString(), její výsledek bude zobrazován v prohlížeči zdrojového kódu.

B.2 Rozšíření animátorů

Animátory jsou velice volné komponenty. Pravidlo je, že by měly držet odkaz na prohlížeč algoritmu a to proto, aby se dostali na prohlížecí vrstvu, kde mohou vkládat své komponenty a připojit posluchače události.

C Přehled operací v jazyce Java

priorita	operátor	operandy	asoc.	popis
13	$++, --$	1	P	pre/post inkrementace/dekrementace
13	$+, -$	1	P	plus, mínus
13	$\sim, !$	1	P	bitový doplněk, logická negace
13	(typ)	1	P	přetypování
12	$*, /, \%$	2	L	násobení, dělení, zbytek po dělení
11	$+, -$	2	L	sčítání, odčítání
10	$<<, >>$	2	L	posun vlevo, posun vpravo
10	$>>>$	2	L	posun vpravo s rozšířením nuly
9	$<, <=$	2	L	menší, menší nebo rovno
9	$>, >=$	2	L	větší, větší nebo rovno
9	instance of	2	L	test instance
8	$==$	2	L	rovno
8	$!=$	2	L	není rovno
7	$\&$	2	L	bitové/logické AND
6	$\hat{}$	2	L	bitové/logické XOR
5	$\vert$	2	L	bitové/logické OR
4	$\&\&$	2	L	podmínkové AND
3	$\vert\vert$	2	L	podmínkové OR
2	? :	3	P	podmínkový operátor
1	$=$	2	P	přiřazení
1	$+=, -=$	2	P	přiřazení s operací
1	$*=, /=$	2	P	přiřazení s operací
1	$<<=, >>=$	2	P	přiřazení s operací
1	$>>>=$	2	P	přiřazení s operací
1	$\&=, \hat{}=, \vert=$	2	P	přiřazení s operací

Tabulka 9: Kompletní přehled operací dostupných v jazyce Java [1]

D Třídní diagram aplikačního rámce

Na následující stránce je vyobrazen třídní diagram celého aplikačního rámce. Mimo tříd a vztahů mezi nimi je zde i načrtnut způsob rozdělení do jednotlivých balíčku. Aplikace využívající framework je vyobrazena úplně nahoře a je naznačeno s kterými třídami musí spolupracovat.